A Final Report
Grant No. N00014-91-J-1369

November 1, 1990 - November 30, 1991

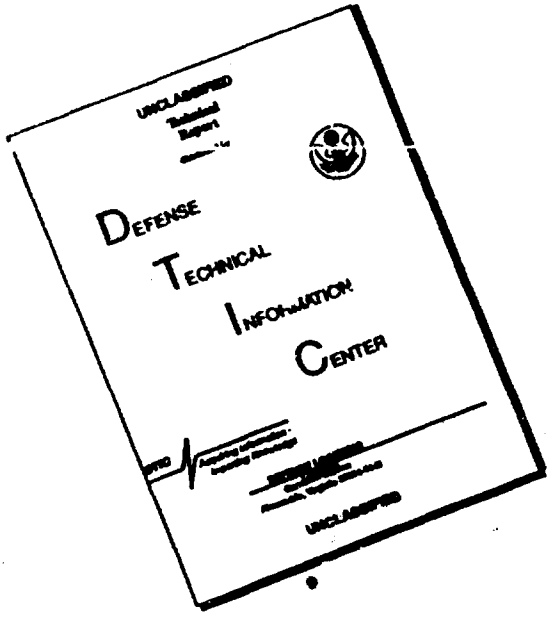
EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS

Submitted to:
Scientific Officer Code: 1133
Andre M. Van Tilborg
Office of Naval Research
800 North Quincy St.
Arlington, VA 22217-5000

Submitted by:

Krithi Ramamritham
Associate Professor
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AD-A246126

A-15

+

A-16

MISSING

FROM ORIGINAL

DOCUMENT

AS RECEIVED

FROM THE

ORIGINATOR

A Final Report
Grant No. N00014-91-J-1369

November 1, 1990 - November 30, 1991

EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS

Submitted to:
Scientific Officer Code: 1133
Andre M. Van Tilborg
Office of Naval Research
800 North Quincy St.
Arlington, VA 22217-5000

Submitted by:

Krithi Ramamritham
Associate Professor
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1992	Final Report 11/1/90 - 1/30/91

4. TITLE AND SUBTITLE
Eighth IEEE Workshop on Real-Time Operating Systems

5. FUNDING NUMBERS
N00014-91-J-1369

6. AUTHOR(S)
Krithi Ramamritham

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Scientific Officer Code: 1133
Andre M. Van Tilborg
Office of Naval Research
800 North Quincy St., Arlington, VA 22217-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This workshop was held May 15-17, 1991 in Atlanta, Georgia. It was the eighth in a continuing series of workshops on real-time operating systems and software and was held in conjunction with the 17th IFAC/IFIP Workshop on Real-time Programming. The workshop was co-sponsored by the IEEE Computer Society Technical Committee on Real-Time Systems and the Office of Naval Research and had as its goals:

- to investigate advances in real-time operating systems, software, and programming languages;
- to promote interaction among researchers and practitioners;
- to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

This report contains the Proceedings of the Workshop as well as the Final Report to IEEE, the IFAC/IFIP Final Report, and additional budgetary information provided to the Office of Naval Research.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	210
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	unlimited

Eighth IEEE Workshop on Real-Time Operating Systems and Software
(in Conjunction with)
IFAC/IFIP Workshop on Real-Time Programming

May 15-17, 1991
Atlanta, GA

In Cooperation with:

•IFAC
Technical Committee on Computers
Working Group in Real-Time Programming

and

•IFIP Working Group 5.4
on Computerized Process Control

Proceedings

Proceedings of the
EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE

(in Conjunction with)

IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING

MAY 15-17, 1991
Atlanta, GA
USA

Workshop Chairs:

Krithi Ramamritham
University of Massachusetts
Dept. of Computer and Information Science

Wolfgang A. Halang
University of Groningen
Dept. of Computing Science

Local Arrangements Chair:

Karsten Schwan
Georgia Institute of Technology

TABLE OF CONTENTS

SESSION I: OPERATING SYSTEMS

SESSION 2: DESIGN OF REAL-TIME SYSTEMS

SESSION 3: APPLICATIONS/EXPERIENCE

SESSION 4: TIMING–ANALYSIS/MONITORING

SESSION 5: POT POURRI

SESSION 6: SCHEDULING POT POURRI

Multiprocessor Synchronization Primitives with Priorities

Evangelos P. Markatos*
markatos@cs.rochester.edu
University of Rochester
Computer Science Department
Rochester, New York 14627

Abstract

Low-level multiprocessor synchronization primitives, such as spinlocks. are usually designed with little or no consideration about timing constraints, which makes them inappropriate for real-time systems. In this paper, we propose a new synchronization mechanism, the *priority spinlock*, that takes into account the priorities of the processes that want to acquire it, and favors high priority processes. We define what a priority spinlock is, and propose two algorithms to implement priority spinlocks with local spinning. Priority spinlocks can be used to provide prioritized *mutually exclusive* access to shared resources in real-time multiprocessor systems. They can also be used as building blocks for higher level priority synchronization primitives, such as priority semaphores.

1 Introduction

Real-time systems have timing constraints that must be met, otherwise catastrophic effects may happen [13]. Timing constraints usually take the form of deadlines, earliest starting times, or value functions. Schedulers translate these timing constraints along with recourse requirements and criticalities into priorities [2, 3, 5, 7, 12], and assign the resources of the system to higher priority processes. These priorities must be preserved throughout the system, otherwise the correctness of a real-time system can not be demonstrated. If there is a server where priorities are not preserved, then a high priority process may be delayed (possibly indefinitely) by a low priority process. This event is called priority inversion, and must be avoided, bounded, or minimized, otherwise the feasibility of the schedule and the correctness of the scheduler implementation are questionable. Our work tries to minimize the priority inversion that can happen in the implementation of the most fundamental synchronization mechanism of a multiprocessor real-time system: the spinlock. Spinlocks are low-level synchronization mechanisms for multiprocessor systems that are used to provide mutual exclusion. They are usually implemented with the help of atomic instructions like `test_and_set` that most multiprocessors provide. However, spinlock implementations are usually proposed with little or no consideration of timing constraints. So, current implementations service requests for the lock randomly, or at best FIFO. Random service of requests, are inappropriate for real-time systems; FIFO service is appropriate in real-time applications, because it bounds the worst case time where a processor has to wait to acquire the spinlock. However, FIFO implementations may force a high priority process that wants to acquire a spinlock to wait for all the low priority processes that happen to have requested the spinlock before it. This potentially long wait leads to very conservative scheduling, because although the worst case priority inversion is bounded, the bound is very large.

In this paper we propose a new version of spinlocks called *priority spinlocks*. Priority spinlocks take into account the priority of the process that wants to acquire the lock, and assign the locks to the highest priority process that has requested the lock at the time the decision is being made.

1.1 Priority spinlocks

Priority spinlocks are defined as follows:

1. No more than one processor may possess a lock at one time.
2. Each processor that competes for a priority spinlock has a *dynamic priority* that reflects the importance of the process it runs. This priority may change over time. All processors have different priorities.
3. Priority spinlocks have the **priority ordering** property.

*This material is based upon work supported by the National Science Foundation under Grant number CDA-8822724

> A spinlock implementation has the priority ordering property iff there is a constant k, such that if a high priority processor requests a lock, then lower priority processors will acquire the lock at most k times before the high priority processor acquires it.

For our proposed algorithms $k = 1$. This means that if more than one processor attempts to acquire the lock, then the highest priority processor acquires it, unless the highest priority processor attempts to acquire the lock *after* it has been decided that the lock should be given to some other lower priority processor.

Our assumptions about the execution model are:

- Processors communicate via a medium (e.g. bus, switch) that has bounded access time. This means that a processor can not be prevented from using the medium indefinitely. If the communication medium does not have bounded access time, there is no way to built a predictable system on top of it [9]. Buses that provide bounded access time include synchronous buses, and asynchronous buses configured with Round-Robin access, like the VME bus [6].

- Processes that request, hold, or release locks are not preemptable during the time they request, hold or release the lock[1].

In this paper we describe two algorithms that implement priority spinlocks. The algorithms are modifications of previous algorithms for FIFO spinlocks by Burns [4] and Mellor-Crummey and Scott [8]. The major properties we add to Burns's algorithm are *priority ordering* of the spinlock and *local spinning*[2]. The major property we add to the MCS lock is priority ordering.

The next section summarizes previous work on hard real-time multiprocessor synchronization primitives, and how it differs from our work. Section three describes our first priority spinlock algorithm that can be efficiently implemented on UMA (Uniform Memory Access) multiprocessors. Section four describes our second priority spinlock algorithm that can be efficiently implemented both in UMA and NUMA (Non Uniform Memory Access) multiprocessors. Section five compares the complexity of the two algorithms in terms of running time, memory requirements, and remote memory references. Finally, section six presents our conclusions.

[1] This assumption is not as restrictive as it sounds, because spinlocks are usually used by the kernel to build high level synchronization primitives. So, the time that a processor holds a spinlock is very small and known beforehand. In addition, non-preemptability of processes that execute spinlock synchronization code enables us to focus on the aspects related to synchronization only and not to preemptability

[2] A spinlock implementation has the property of local spinning if each processor spins on a local memory location, or on a local cache copy, without generating any remote memory references or bus accesses. Remote memory accesses can easily saturate the bus (or switch) and slow down the whole system. Local spinning is essential for the efficient and predictable behavior of real-time systems.

2 Previous work

Synchronization in multiprocessor hard real-time systems is a relatively new field. Molesky, Shen and Zlokapa [9], describe predictable algorithms for semaphores with linear waiting. Although their proposed algorithms are predictable, they do not take into account the priorities of the processes that want to acquire the semaphore.

Rajkumar, Sha and Lehoczky [11] presented a multiprocessor extension of the priority ceiling protocol [10]. The priority ceiling protocol is a protocol that minimizes priority inversion for a set of periodic real-time processes that access exclusively some shared data. The multiprocessor priority ceiling protocol generalizes the uniprocessor priority ceiling protocol by executing all the critical regions associated with a semaphore on a particular processor (called *synchronization processor*). So, the critical regions in the programs are substituted by an invocation to a remote server that deals with all the critical regions associated with a particular semaphore. Remote invocation is an attractive alternative to mutual exclusion algorithms, especially in multiprocessors where the cost of accessing non-local memory is very high. However, the existence of a remote centralized server limits the scalability of the solution, and increases the cost of executing fine grain sharing applications.

Anderson [1], and Mellor-Crummey and Scott[8] derived spinlock implementations that service lock requests in FIFO order and can be used in real-time systems that just want to bound (not minimize) the priority inversion.

Our work focuses on how to built prioritized low-level synchronization primitives. In this respect, it is mostly related to [9], where they describe how to build predictable FIFO low-level synchronization primitives for real-time multiprocessor systems.

2.1 Our approach

Mutual exclusion algorithms for multiprocessors exist for quite a while now. All these algorithms focus on providing mutual exclusion, along with one or more *desired* properties that a multiprocessor mutual exclusion mechanism should have, such as *starvation* avoidance, *fairness*, FIFO service of lock requests, *local spinning*, etc. However, none of the previous multiprocessor mutual

exclusion algorithms has considered properties such as *timeliness* or *priority ordering*, which are vital for real-time systems. So, previous mutual exclusion mechanisms service requests for the lock independently of the timing constraints of the processes that requests the lock. In this report we focus on mutual exclusion algorithms that consider the timing constraints of the processes that request it. The timing constraints should be expressed in the form of priorities[3], so that the mutual exclusion implementation, will be able to service the requests for the lock in priority order.

3 First algorithm

This section presents an algorithm for acquiring and releasing priority spinlocks that can be efficiently implemented on cache-coherent multiprocessors.

3.1 Burns's algorithm

Our algorithm is a modification of Burns's algorithm [4], that guarantees mutual exclusion and *linear waiting*. Linear waiting means that if a processor tries to enter the critical region, it will enter it before any other processor has entered the critical region more than once.

Burns's algorithm uses a shared global array `want[0..P-1]`, where `P` is the number of processors in the system. If processor `i` wants to enter the critical region, it sets `want[i]` to `TRUE`. Then, it starts spinning executing the `test_and_set` atomic instruction and checking the value of `want[i]`.

A processor `i` that exits the critical region searches the array `want` sequentially, finds the first spinning processor with index larger than the index of processor `i` (modulo the number of processors), and allows it to enter the critical region. Burns's algorithm can be found in figure 1.

```
local int i ; // the id of this processor
boolean array want[0..P-1] ;
initially for all k: want[k] = FALSE;

acquire_lock(l)
{
   want[i] = TRUE;//register that I want the lock
   while ((want[i] == TRUE)  AND
     (test_and_set(l) == locked)) ;// spin
}
release_lock(l)
{
   int j ;
   want[i] = FALSE ; // release the lock
   // get the next processor in line
   j = (i+1) % P ;
   while ((want[j] == FALSE)  AND (j != i))
     // while no processor wants the lock
     //  go to the next one
     j = (j+1) % P ;
   if (i == j)  // if none wants the lock
     clear(l) ;  // release it
   else // hand  processor j  the lock
     want[j] = FALSE ;
```

Figure 1. Burns's FIFO lock

3.2 Adding priorities to Burns's lock

Our first algorithm is a straightforward modification of Burns's algorithm and achieves the following properties:

1. **Priority ordering** : The processor that releases the lock sequentially scans a shared array that contains the identity of all the processors that currently spin for this lock, and finds the highest priority spinning processor. Then, it gives the lock to this processor.

2. **Local spinning** : In Burns's algorithm processors continually execute the `test_and_set` atomic instruction and poll a (potentially) remote memory location. We modify the algorithm in such a way that if implemented on a multiprocessor with *coherent caches*, then processors spin only on local cache copies. The spinning code in Burns's algorithm is:

```
while((want[i] == TRUE)  AND
      (test_and_set(l) == locked));
```

Our spinning code is:

```
while ((want[i] == TRUE) AND
       (l == locked OR
        test_and_set(l) == locked));
```

Note that if the value of lock `l` is coherently cached, then all the processors will spin on local copies for as long as the lock is held. Moreover, we free the lock in such a way, that when the lock is given from one processor to another, the spinning processors do not cache miss, and so they do not generate any bus or switch traffic.

The code of our algorithm is in figure 2.

3.2.1 Description

For each spinlock there is an array `want[0..P-1]` that represents the desire of each processor to use the spinlock.

[3]The priorities may represent deadlines, or periods, or some other form of timing constraints.

Acquire lock: If processor `i` wants to acquire the lock then it sets `lock(i)` to `TRUE`, and registers its priority (found in local variable `my_priority`) in the shared array `priority[0..P-1]`. Then, it starts spinning waiting for the lock.

Release lock: When the spinlock is released, the array `want` is sequentially scanned to find the highest priority processor currently spinning to give it the lock. The spinlock is given from processor `i` to processor `j` by negating the spinning variable `want[j]` on which processor `j` spins.

```
boolean array want[0..P-1] ;
boolean array priority[0..P-1] ;
initially for all k: want[k] = FALSE;
                     priority[k] = 0 ;
local int my_priority ;
local int i ; // processor's id

procedure acquire_lock(L)
   // i is the identity  of the processor
   priority[i] = my_priority ;
   want[i] = TRUE ;
   while ((want[i] == TRUE) AND
        (L == locked OR
        test_and_set(L) == locked))  ;
        // while the spinlock is held spin

procedure release_spinlock(L)
   want[i] = FALSE ;
   j = find_max_priority() ;
   // find the highest priority spinning processor
   if (j == -1) {//if no processor spins
     L = free ;  // clear the spinlock
   }
   else { // if there is a spinning processor
     // allow it to continue, without releasing
     // the spinlock;   the spinlock is inherited
     // by  processor j from processor i
     want[j] = FALSE ;
   }
```

Figure 2: Priority spinlock with local cache spinning

4 Second algorithm

The efficient implementation of the previous algorithm requires a multiprocessor with coherent caches. Multiprocessors with coherent caches do not scale to large numbers of processors due to bus bandwidth limitations. Multiprocessors without coherent caches are widespread and come in configurations with hundreds of processors. The algorithm we describe in this section works equally well on multiprocessors with coherent caches or with local memories. The priority spinlock algorithm we propose is a modified version of the MCS lock [8]. Our algorithm and the MCS lock spin on local memory only[4], require a small constant amount of space per lock, and work equally well on machines with and without coherent caches. Moreover our algorithm guarantees the *Priority ordering* property.

4.1 The MCS lock

The MCS lock uses a queue of processors for each lock. Processors that want to acquire the lock join the queue. The order in which processors join the queue is the order in which they acquire the lock. The lock is nothing more than a pointer to the head of the queue. The tail of the queue represents the processor that has the lock.

Acquire lock: If processor `proc` wants to acquire the lock, it joins the queue by executing a `fetch_and_store`[5] atomic instruction. If `proc` is the tail of the queue (no predecessor), then it continues (because the tail of the queue is the holder of the lock), else it busy-waits by spinning on a local variable. The pseudocode to acquire the lock is:

```
predecessor = fetch_and_store(lock, my_record) ;
    // my_record now is a pointer to the head
    // of the queue, and the lock pointer now
    // points to my_record
if  (predecessor == nil)//I am the holder
    return ;
else
    // spin on some local variable
```

Release lock: When processor `proc` wants to release the lock then:

- If the lock queue is not empty, it gives the lock to its successor, by changing the value of its successor's spinning variable.
- If the queue is empty, then the code is more complicated. If processor `proc` releases the lock after it verifies that the queue is empty, then there is the possibility that in the meantime (after verifying, but before releasing) some other processor will join the queue and will stay there forever. In order to avoid this deadlock situation the releasing processor (`proc`) uses the atomic instruction `compare_and_swap`[6]: Processor `proc` checks if it is the head of the queue, and if yes it dequeues itself.

[4] More precisely, on statically allocated processor specific memory, which will be local to any machine in which shared memory is distributed or coherently cached.

[5] `fetch_and_store(x,y)` atomically sets `x` to `y` and returns the old value of `x`

[6] `compare_and_swap(x,y,z)` atomically compares `x` and `y` and if equal, sets `x` to `z` and returns `TRUE`, else it returns `FALSE`.

The check and dequeuing are done atomically with the help of the atomic `compare_and_swap` instruction. If the atomic instruction doesn't succeed, it means that some other processor joined the queue. In this case processor `proc` gives the lock to the processor that just joined the queue. If more than one joined it, the first one gets the lock.

The pseudocode for the algorithm is in figure 3.

```
struct queue_link {
    struct lock_struct * next ;
    Boolean locked ;
}
typedef struct queue_link * qlink ;

local qlink I  ;
    // initialized to point to a queue link record
    // in the local portion of shared memory

procedure acquire_lock (L)
    qlink  predecessor  ;
    I->next = nil ;
    predecessor = fetch_and_store (L, I)
    if (predecessor != nil){   // queue was non-empty
        I->locked = true ;
        predecessor->next = I ,
        repeat while (I->locked == TRUE) ;// spin
    }

procedure release_lock (L)
    if (I->next == nil) { // no known successor
        if compare_and_swap (L, I, nil)
            return ;
            // assuming compare_and_swap
            // returns true iff it swapped
        repeat while (I->next == nil) ; // spin
    }
    I->next->locked = FALSE ;
```

Figure 3: The MCS list-based queuing lock

4.2 Priority lock

Our priority spinlock is very similar to the MCS lock. The basic difference is that we dequeue the processors from the lock queue in a priority order, because we want to implement priority spinlocks that have the priority ordering property. Dequeuing the processors from the queue in priority order is not trivial because the queue can be accessed by any processor that wants to join it, and we need to keep it in consistent state. Of course, we can not use locks for the enqueue and dequeue operations, because these locks can only be simple (not priority) spinlocks, and they may violate the priority ordering property. So, we should protect the queue using only atomic instructions.

Data structures: We use a global queue of processors that spin for the lock. The queue is implemented as a doubly linked list. Each processor has a record in the queue that contains some information about the processor. The lock is a pointer to the head of the queue. If the queue is empty, then the lock is free. If the queue is not empty, then the holder of the lock is the tail of the queue. The global definitions for the extended MCS lock are in figure 4.

Acquire lock: The `acquire_lock` operation is not changed. If a processor wants to acquire the lock it joins the lock queue atomically, using the help of the `fetch_and_store` atomic instruction. If it finds out that it is the tail of the queue, then it continues, knowing that it has the lock. Otherwise, it spins on a local variable, waiting its turn to become the tail of the queue and the holder of the lock. The code for the `acquire_lock` operation is in figure 4.

```
struct queue_link{
    struct queue_link * next ;
    struct queue_link * previous  ;
    int priority ;
    Boolean  locked ;
}
typedef struct queue_link * qlink  ;

qlink I  ;
  // initialized to point to a queue link record in
  // the local portion of shared memory

procedure acquire_lock (L)
{
    qlink  predecessor  ;
    I->next = nil ;    // initialize the fields
    I->previous = nil ;// of the record
    I->priority = my_priority ;
    predecessor = fetch_and_store (L, I) ; // join
    if (predecessor != nil) { //queue was non-empty
        I->locked = TRUE ;   //the lock is locked
        I->previous = predecessor ;//join the queue
        predecessor->next = I ;
        repeat while (I->locked == TRUE)  ;  // spin
    }
}
```

Figure 4. Definitions and `acquire_lock` procedure for the second priority spinlock algorithm (constant space and local spinning both on NUMA and UMA machines).

Release lock: The processor that releases the lock searches the queue to find the highest priority spinning processor and gives it the lock. The action of giving the lock from processor A to processor B is done by moving processor B to the end of the queue, and removing processor A from the queue. This is not a trivial task to do,

because at the time processor B is being moved to the end of the queue, other processors may join the queue, and race conditions may happen. Note that we can not use a simple (not priority) lock to protect the queue, because this lock may introduce priority inversion that may violate the priority ordering property. During the `release_lock` operation there are many cases to be considered. Depending on where in the queue is the highest priority processor, we have to execute different code to move it to the end of the queue.

Define h to be the highest priority processor in the queue. Define I to be the processor that releases the lock. The different cases to be considered are:

- *I is the only processor in the queue:* In this case I will release the lock. However, it has to make sure that no processor joined the queue after I found that it is the only processor in the queue, and before releasing the lock. In the case where some processor did join the queue between the two events, then processor I should give the lock to this processor[7].

```
if (I->next == nil) { // no one in the queue
    if compare_and_swap (L, I, nil) return ;
    // compare_and_swap returns true iff swapped
    // some processor just joined the queue
    repeat while I->next = nil  ;  // spin
    // give the lock to the first processor
    I->next->locked = false ;
    return ;
} // else there are spinning processors
```

- *Processor h is just after processor I in the queue:* In this case there is no need to move h to the end of the queue, because once I is removed from the queue, then h will be the last processor in the queue, and the holder of the lock. The code to do this is:

```
h = highest_priority_spinning_processor() ;
if (I->next == h){
    //the queue is: I -> h -> ...  <- L
    h->previous = nil ; // dequeue I
    // give the lock to processor h
    h->locked = FALSE ;
    return ;
}
```

- *Processor h is neither penultimate, nor first in the queue:* In this case we can move h to the tail of the queue, without any possibility of race condition, because processors join the queue through its head, and alter only the pointers of the processor that is the head of the queue. The code in this case is:

```
// the queue is : I->... ->h->...<-L
p = h->previous ;
if (h->next != nil) {
    // the queue: I->...p->h->...<-L
    // remove h from the Q
    h->previous->next = h->next ;
    h->next->previous = h->previous ;
    // put h at the end of the Q
    h->next = I->next ; h->locked = FALSE ;
    h->previous = nil ;
    return ;
}
```

- *Processor h is the head of the queue:* This case is more difficult than the previous ones, because we have to move h to the end of the queue, while at the same time other processors may join the queue, and change the pointers of the record of h. Again we use the help of the `compare_and_swap` atomic instruction. Define p to be the predecessor of h. We will use the `compare_and_swap` to move p to the head of the queue, and move h to the tail of the queue without any race conditions. We atomically do the following:

  if h is the head of the queue then set p to be the head of the queue

Now, if the above atomic operation succeeded, then p is the head of the queue, and h can be safely moved without any race conditions. Otherwise, it means that one or more processors have just join the queue, and in this case we can move h to the tail of the queue safely. The pseudocode to do this is:

```
p->next = nil ;
if (compare_and_swap(L, h, p)) {
  // the swap was successful
  // after the compare_and_swap the
  // queue becomes  I->...->p<- L
  // p is now the head of the queue
  //  move h to the tail
  h->next = I->next ; h->previous = nil ;
  h->locked = FALSE ; return ;
} else {
  // some other processor just joined
  // compare and swap did not return true
  repeat while h->next = nil ;  // spin
  h->previous->next = h->next ;
  h->next->previous = h->previous ;
  h->next = I->next ; h->previous = nil ;
  h->locked = FALSE ;
}
```

[7] Note that more than one processors, may have joined the queue in the meantime. The lock is given to the first of them without considering priorities. The highest priority processor will acquire the lock the next time, which is consistent with our definition of priority spinlocks.

5 Complexity

We describe the performance of our priority spinlock algorithms in terms of running time, memory require-

ments, and remote memory references. Both the algorithms have the same asymptotic complexity in terms of running time and remote memory references (or cache invalidations). Suppose that at most P processors want to lock the spinlock i. The time to acquire a lock may be unbounded, because there may always be high priority processors in the systems that request the lock, and some low priority processor may never get the chance to acquire it. The time to release the lock is $O(P)$. The number of remote memory references (or cache invalidations), to acquire a lock is constant, and $O(P)$ references to release it. The memory requirements of the two algorithms are different. If there are P processors in the system and M locks, then the total memory required by the prioritized MCS lock is $O(P + M)$, while the prioritized Burns's lock is $O(P \cdot M)$.

6 Conclusions

In this paper we defined a low level multiprocessor synchronization mechanism called priority spinlock, and we proposed two algorithms that implement priority spinlocks. Priority spinlocks take into account the timing constraints of the processes that want to acquire the lock, and favor high priority processes. Priority spinlocks can be used in many places in a real-time application. They can be used as described to provide prioritized access to a shared resource, or as a building block for priority binary semaphores, priority counting semaphores, and priority monitors. Our algorithms have predictable implementations because they are based on local spinning. The first can be used in multiprocessors with coherent caches, and the second, albeit more complicated, can be used in multiprocessors with and without coherent caches.

7 Acknowledgments

I would like to thank Tom LeBlanc, Michael Scott and John Mellor-Crummey for valuable discussions and suggestions in earlier drafts of this paper.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] S.R. Biyabani, J.A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *IEEE Real-Time Systems Symposium*, pages 152–160, 1988.

[3] J. Blazewicz. Deadline scheduling of tasks with ready times and resource constraints. *Information Processing Letters*, 8(2):60-63, 1979.

[4] J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2), Summer 1978.

[5] S.C. Cheng, J.A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. *IEEE Tutorial on Real-Time Systems*, pages 150–173, 1988.

[6] Motorola Inc. MVME135, MVME135-1, MVME135A, MVME136 and MVME136A 32-bit microcomputers user's manual, 1989.

[7] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[8] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, to appear. Earlier version published as TR 342, University of Rochester, Computer Science Department, Aprir 1990.

[9] L.D. Molesky, C. Shen, and G. Zlokapa. Predictable synchronization mechanisms for multiprocessor real-time systems. *Journal of Real-Time Systems*, 3(2), 1990.

[10] R. Rajkumar, L. Sha, and J.P. Lehocsky. An experimental investigation of synchronization protocols. In *Proceedings 6th IEEE Workshop on Real-time Operating Systems and Software*, pages 11–17, Pittsburgh, PA, May 1989.

[11] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–169. 1988.

[12] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the 7th Real-Time Systems Symposium*, pages 181–191, New Orleans, LA, 1986.

[13] J.A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10-19, Oct 1988.

YARTOS: Kernel support for efficient, predictable real-time systems

Kevin Jeffay, Don Stone, Dan Poirier
University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175

January 1991

1. Introduction

Real-time computer systems are loosely defined as the class of computer systems that must perform computations and I/O operations in a time frame defined by processes in the environment external to the computer. Real-time systems differ from more traditional multiprogrammed systems in that real-time systems have a dual notion of correctness. In addition to being logically correct, *i.e.*, producing the correct outputs, real-time systems must also be temporally correct, *i.e.*, produce the correct output at the correct time. In this paper we describe an operating system kernel called YARTOS (Yet Another Real-Time Operating System) that supports the construction of efficient, predictable real-time systems. Initially we are focusing on the problem of designing and constructing *hard-real-time* systems. Hard-real-time systems are real-time systems that require deterministic guarantees of temporal correctness. These are systems in which the cost of failing to interact with the external environment in real-time is high. This high cost can be measured in monetary terms (*e.g.*, an inefficient use of raw materials in a process control system), aesthetic terms (*e.g.*, unrealistic output from a computer music or computer animation system), or possibly in human or environmental terms (*e.g.*, an accident due to untimely control in a nuclear power plant or fly-by-wire avionics system).

In recent years, numerous real-time operating systems have been developed. Our work is distinguished by the programming model that YARTOS supports and by the aggressive use in YARTOS of recent results in the theory of deterministic processor and resource allocation. The programming model supported by YARTOS is an extension of Wirth's discipline of real-time programming [Wirth 77]. In essence it is a message passing system with a semantics of inter-process communication that specifies the real-time response that an operating system must provide to a message receiver. This allows us to assert an upper bound on the time to receipt and processing of each message. The exact response time requirement is a function of such factors as the rate with which a process receives messages on a given channel. Ultimately, these rates are functions of the rates at which data arrives from external sources. These semantics provide a framework both for expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. The programming model is described in greater detail elsewhere [Jeffay 89a].

Programs in YARTOS are compiled into a set of *sporadic* tasks that share a set of serially reusable, single-unit resources. A sporadic task is a sequential program that is invoked in response to the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (*e.g.*, an interrupt from a device) or by processes internal to the system (*e.g.*, the arrival of a message). We assume events are generated repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. A resource in YARTOS is a software object (*e.g.*, an abstract data type) that is

shared (read/write) by multiple tasks. For a given workload, the goal of YARTOS is to guarantee that (1) all requests of all tasks will complete execution before their deadlines and (2) no shared resource is accessed simultaneously by more than one task. We have developed an optimal (preemptive) algorithm for sequencing such tasks on a single processor [Jeffay 89b, 90]. The algorithm is optimal in the sense that it can provide the two guarantees whenever it is possible to do so. Moreover, an efficient algorithm has been developed for determining if a workload can be guaranteed a correct execution [Jeffay 90]. Our development and analysis of a formal scheduling model has resulted in a surprisingly efficient implementation of YARTOS tasking. Specifically, applications consisting of multiple tasks can be executed on a single run-time stack and no explicit locking mechanism is required for accessing shared resources. This improves the memory utilization of the system and yields efficient context switches between tasks. This encourages liberal use of tasks and data sharing in YARTOS applications.

In this note we concentrate on the YARTOS's scheduling model and its implementation. The following section describes the scheduling model and the algorithms used for processor and resource allocation. Section three briefly describes a prototype implementation of the YARTOS kernel. We conclude with some brief comments on our experiences with YARTOS.

2. Scheduling Model

YARTOS supports two basic abstractions: *tasks* and *resources*. A task is an independent thread of control that is invoked at sporadic intervals. Each invocation of a task must complete execution before a well-defined deadline. The invocation intervals and deadlines for a task are derived from constructs in the higher-level programming model. During the course of execution, a task may require access to some number of *resources*. A resource is a software object (an abstract data type) that encapsulates shared data and exports a set of procedures for accessing and manipulating the data. Like a monitor, objects require mutually exclusive access to the data they encapsulate. A set of tasks is said to be *feasible* if all requests of all tasks will complete execution before their deadlines and no shared resources is accessed simultaneously by more than one task.

We have developed an efficient decision procedure for deciding if a set of tasks is feasible. Let $r_1, r_2, ..., r_n$ be the rates at which tasks are invoked (measured in terms of minimum inter-invocation time), sorted in non-increasing order. Let $C_1, C_2, ..., C_n$ be the maximum execution times required to execute the tasks. Let n_i be the number of operations on shared resources repositories performed by an invocation of task i, and let $c_{i1}, c_{i2}, ..., c_{in_i}$ be the maximum execution time required for each operation. Let c_{i0} be the maximum execution time required to execute the remaining code (sequential code in task i that does not require access to a shared resource). Hence $C_i = c_{i0} + c_{i1} + c_{i2} + ... + c_{in_i}$. A set of tasks will be feasible on a single processor if:[1]

---

[1] Necessary and sufficient conditions for schedulability are presented in [Jeffay 90]. For brevity, we present a simpler (sufficient) formulation of these conditions.

1) $\sum_{i=1}^{n} C_i r_i \leq 1,$

2) $\forall i, 1 < i \leq n; \forall k, 1 \leq k \leq n_i; \forall L, 1/r_1 < L < 1/r_i :$

$$L \geq c_{ik} + \sum_{j=1}^{i-1} \lfloor (L-1)\, r_j \rfloor C_j ,$$

The product $C_i r_i$ is the fraction of the processor that must be allocated to processing invocations of task i. The first condition stipulates that the processor not be overloaded. Condition (2) applies to tasks that require access to resources (tasks for which $n_i > 0$). It quantifies the processor demand that occurs when tasks simultaneously access a shared resource. The right hand side of the inequality in condition (2) is a least upper bound on the processor demand that can be realized in an interval of length L starting at the time an invocation of a resource requesting task i is scheduled, and ending sometime before the deadline for completion of the invocation. Under all circumstances this bound must be less than or equal to the minimum inter-invocation time (or a fraction thereof) of task i. A set of tasks can be tested against these conditions in time $O(1/r_n)$.

The derivation of the feasibility conditions has suggested an algorithm for sequencing the tasks. The algorithm is a variation of the well-known *earliest deadline first* (*EDF*) scheduling algorithm; a preemptive priority driven scheduling algorithm with dynamic priority assignment [Liu & Layland 73]. The novel feature of the algorithm is the fact that it dynamically manipulates the deadline of a of task invocation to ensure that the task maintains exclusive access to whatever shared resource it might be accessing. This is similar to the concept of a priority ceiling in priority inheritance protocols [Sha et al. 90]. Our approach is new because (1) it is optimal, and (2) it is based on EDF scheduling. This manipulation of deadlines ensures that there will exist no contention for shared resources at run-time. Because of this YARTOS need not provide any special locking facilities for shared resources.

A second interesting property of our scheduler concerns the implementation of tasks. If a task P is preempted, it is the case that any task that executes while P is preempted, is guaranteed to complete execution before P is resumed. Since tasks execute to completion, we may execute all tasks on a single run-time stack. This greatly improves memory utilization and reduces context switching overhead. This is similar to Baker's stack allocation policy [Baker 90]. These two properties of our scheduling algorithm affords us an extremely efficient implementation of tasks.

3. Implementation

A YARTOS prototype that implements both the tasking model and scheduling discipline has been constructed. It is implemented in *C* on an IBM PS/2 computer (Intel 80386 processor) and consists of approximately 1500 lines of code. Its small size is due largely to the fact that we may implement tasks and resources in a simple and straightforward manner. YARTOS is being used to experiment with digital audio and video on a local area network of workstations [Jeffay & Smith 90a, 90b]. Currently we have small number of workstations interconnected with a 16Mbit token ring network. Each workstation is connected to a video camera, microphone, speaker, and video monitor, and contains real-time video encoding/decoding hardware with compression capabilities (Intel's ActionMedia Digital Video Interactive product [Ripley 89]). The goal is to use the workstation and

network to emulate a cable television network. This requires real-time data acquisition, processing and transmission. The motivation for this project is to support the remote, real-time collaboration of scientific and technical professional [Smith et al. 90].

Previously, YARTOS was used to successfully support an interactive 3-dimensional graphics display system used for research in *virtual worlds* [Chung et al. 89]. The graphics system is a head-mounted display system consisting of a helmet with miniature television monitors embedded in it, and tracking hardware for the helmet and for a hand-held pointing device. A computer generated image of a 3-dimensional "virtual world" is displayed in the helmet. The goal of the system is to track the user's head and pointing device in real-time and to update the image displayed in the helmet so as to maintain the illusion that the user is immersed in an artificial world. There are two separate real-time concerns in this application. First, the system must update the display at a rate sufficient for ensuring that animate objects displayed in the helmet move in a smooth and realistic manner. Second, as the user moves her head or pointing device, the displayed image must appear to move with the user's movements. Such constraints were simple to model and realize with YARTOS.

YARTOS and its programming system provide a framework for both expressing processor-time-dependent computations and for reasoning about the real-time behavior of programs. For example, for our implementation of YARTOS, we can demonstrate that the maximum time between the arrival of a complete head and hand position report and the display of an image based on the new position information is approximately 100ms. By performing some simple restructuring of the program, we were able to reduce this performance guarantee to approximately 33ms (see [Jeffay 89a]).

4. References

[Baker 90] Baker, T.P., *A Stack-Based Resource Allocation Policy for Real-Time Processes*, Proc. Eleventh IEEE Real-Time Systems Symp., Orlando, FL, December 1990, to appear.

[Chung et al. 89] Chung, J.C., Haris, M.R., Brooks, F.P., Fuchs, H., Kelley, M.T., Hughes, J., Ouh-young, M., Cheung, C., Holloway, R.L., Pique, M., *Exploring Virtual Worlds with Head-Mounted Displays*, Non-Holographic True 3-Dimensional Display Technologies, SPIE Proceedings, Vol. 1083, Los Angeles, CA, January 1989.

[Jeffay 89a] Jeffay, K., *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15, September 1989.

[Jeffay 89b] Jeffay, K., *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, Proc. Tenth IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 295-305.

[Jeffay 90] Jeffay, K., *Scheduling Sporadic Tasks With Shared Resources in Hard-Real-Time Systems*, University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-038, August 1990. (Submitted for publication.)

[Jeffay & Smith 90a]
Jeffay, K., Smith, F.D., *Designing a Workstation-Based Conferencing System Using the Real-Time Producer/Consumer Paradigm*, Proc. of the First Intl. Workshop on Network and Operating System Support for Digital Audio and Video, Intl. Computer Science Institute, Berkeley, CA, November 1990.

[Jeffay & Smith 90b]
Jeffay, K., Smith, F.D., *System Design for Workstation-Based Conferencing With Digital Audio and Video*, University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report, October 1990. (To appear.)

[Liu & Layland 73]
Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, **Journal of the ACM**, Vol. 20, No. 1, (January 1973), pp. 46-61.

[Ripley 89] Ripley, G.D., *DVI - A Digital Multimedia Technology*, **CACM**, Vol. 32, No. 7, (July 1989), pp. 811-822.

[Sha et al. 90] Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, **IEEE Trans. on Computers**, Vol. 39, No. 9, (September 1990), pp. 1175-1185.

[Smith et al. 90]
Smith, J.B., Smith, F.D., Calingaert, P., Hayes, J.R., Holland, D., Jeffay, K., Lansman, L., *UNC Collaboratory Project: Overview*, University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-042, November 1990.

[Wirth 77] Wirth, N., *Toward a discipline of real-time programming*, **CACM**, Vol. 20, No. 8 (Aug. 1977), 577-583.

Dynamic Scheduling for Hard Real-Time Systems: Toward Real-Time Threads

Hongyi Zhou Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

1 Introduction

Our experiences with real-time and parallel architectures[25], applications[15], and operating systems[21. 24, 7] have led to the conclusion that the requirements for real-time software may change during the lifetime or even during the operation of the system being controlled by such software[5, 15, 13, 12]. We call such systems *dynamic* when their execution environments exhibit potentially unpredictable behavior. For example, their control software may have to contend with (1) high variations in the rates with which external events occur, or (2) a combinatorially large number of possible, joint event occurrences, or (3) variable requirements in reactions to such events (e.g., multiple operating modes)[21], or (4) unexpected changes in the computing resources (e.g., increasing error rates[2, 3], unexpected events[7], or dynamic performance/reliability tradeoffs[1]).

This research addresses the **timeliness** of thread execution in dynamic real-time systems. Specifically, we investigate the dynamic scheduling of execution threads with well-defined timing constraints. Like most researchers in real-time systems[16, 29], we assume that such timing constraints are described by hard deadlines, which must be met in order for thread execution to be useful (e.g., to avoid catastrophic failures). Each individual thread has a well-defined maximum execution time, start time (which need not be equal to its arrival time), and deadline, as well as precedence constraints with respect to other threads.

We have developed a dynamic optimal uniprocessor scheduling algorithm with an $O(n \log n)$ worst case complexity. The algorithm permits thread preemption and is built upon the earliest-deadline-first scheduling policy shown optimal in [6]. However, the presented algorithm has higher efficiency than other known algorithms[6, 9, 22], and it also allows threads to have arbitrary start times (which need not be equal to their arrival times), deadlines, and precedence constraints. Furthermore and in contrast to [6], our dynamic algorithm not only selects the next thread to be run at the time of a context switch, but also performs schedulability analysis and accepts or rejects a thread at the time of its creation.

High efficiency of our algorithm is achieved by the efficient representation of the scheduling information. Specifically and in contrast to other work in real-time scheduling[29, 22], schedulability analysis does not rely on an explicit representation of the information regarding the assignment of thread executions to time periods. Instead, we use a data structure termed a *slot list* to record only the time periods at which threads have been scheduled. The selection of threads for execution by the low-level thread multiplexor is performed using an additional data structure: an earliest deadline list termed the *EL*. As in [9] and in contrast to [22], threads are run by the earliest deadline and start time.

The dynamic scheduling algorithm constitutes the necessary basis for operating systems that address dynamic real-time systems. We have embedded the algorithm in a real-time threads library implemented on a 32-node BBN Butterfly multiprocessor. This library provides a portable, efficient basis for a family of real-time, multiprocessor operating system kernels[19, 23] that jointly support highly dependable, parallel and real-time computing[7, 2]. The algorithm's embedding within the real-time threads library permits its experimental evaluation. As a result, the analytic model capturing the algorithm's average case behavior can be validated with actual system measurements. Specifically, since the $O(n \log n)$ worst case complexity

of the algorithm depends on the number of slots in the slot list, which tends to decrease with increasing system load, the algorithm's average case performance should be much better than $O(n \log n)$. Both theoretical performance analysis and measurements of the algorithm's implementation on the BBN Butterfly multiprocessor support this conjecture [26].

2 Definitions and Related Research

We consider the problem of scheduling a set of n preemptable threads on a uniprocessor system. Each thread is characterized by (A, S, C, D), where A is its arrival time, S is the earliest possible time at which its execution may begin, C is the maximum computation time, and D is the deadline, which are assumed known when the thread arrives at the system. A thread is *preemptable* if its C_i units of computation time can be satisfied by one or more time slots which sum to C_i.

Given a set of threads, a schedule is *feasible* if all the threads in the set can be scheduled such that their timing and precedence constraints are met. A set of threads is *schedulable* if there exists at least one algorithm that can feasibly schedule the set. A static scheduling algorithm is said to be *optimal* if, for any set of threads, it finds a feasible schedule whenever any other algorithm can do so. In dynamic systems, this definition cannot be used, since threads may arrive randomly and since all their characteristics may not be known a priori. In this case, a newly arriving thread is said *schedulable*, if all previously scheduled threads remain schedulable and if the new thread can also be scheduled. Then, a dynamic algorithm is *optimal* if any thread determined unschedulable by the algorithm also cannot be scheduled by any other algorithm.

For the problems defined above, two types of uniprocessor schedulers have been developed in the past. The *rate monotonic* scheduling algorithm developed by Liu and Layland [14] determines thread schedulability for periodic threads. Several extensions of that algorithm deal with sporadic threads[18, 10]. However, in all of their approaches, periodic threads are given higher priorities than sporadic threads. This does not suffice for the dynamic systems we are considering, in which sporadic threads may handle exceptional and important events that may exhibit hard deadlines (e.g., a robot vehicle stumbling or dealing with sudden cross-winds).

The other type of uniprocessor schedulers developed in the past are based on *the earliest-deadline-first* scheduling algorithm, which was shown optimal by Dertouzos[6] for scheduling preemptable threads with arbitrary arrival times. This algorithm can be used for thread scheduling, but not for thread schedulability analysis. Moreover, a limitation of the approach is that each thread's arrival time is assumed equal to its start time. This is not a useful assumption in highly dynamic systems, where threads which handle exceptions may be pre-created (pre-forked)[7] to reduce the latency of exception handling. Similarly, we wish to guarantee the scheduling of groups of threads, in each of which all of the threads have the same arrival time but different start times.

Horn [9] once developed an $O(n^2)$ static algorithm to schedule threads with arbitrary start times and deadlines. It was shown that the maximum lateness, for the single machine case, is minimized by executing all threads in order of increasing deadline.

3 A Dynamic Optimal Scheduling Algorithm

In this section, a brief outline of the dynamic scheduling algorithm is presented, followed by a summary of the interesting results and extensions. Details can be found in [26].

Scheduling algorithm. In the following, we first assume all threads are preemptable and mutually independent. According to the earliest-deadline-first scheduling policy, threads are scheduled earliest deadline first and as close to their start times as possible. Specifically, the scheduling information used by the algorithm is recorded in a *slot list*. Each element of the list represents a time slot already assigned to the threads.

To test a thread's schedulability, the algorithm searches the slot list for the available time intervals between slots. Such search starts at a slot compatible with the thread's start time and ends at a slot compatible with the thread's deadline or when the accumulated length of available time slots is equal to the thread's execution time. The thread is schedulable if sufficient execution time is found during this search, else the algorithm reports the thread as unschedulable. After a thread is scheduled, the slot list is updated. When the number of slots is large, slots are indexed by a balanced binary tree on their start times. Therefore, a slot with a particular start time can be located in the slot list by searching the binary tree in $O(logn)$ time.

The resulting thread to slot assignment is not recorded within the slot data structure. Instead, once a thread's schedulability is determined, it is entered in order into an earliest deadline first list – abbreviated EL. This list records all threads that have previously been determined schedulable. This is the list from which the multiplexor selects a thread for execution when it is ready to run a next thread. The EL is a list instead of a queue, because the thread chosen to run next is not necessarily the first one in the list. This is because the thread with the earliest deadline may not yet have reached its start time. Thus, it is necessary to search the EL for the first 'ready' thread. A thread is ready to run when its start time is earlier than or equal to the current time. If such a search chooses the i-th thread T_i, for example, this means that the start times of all threads before T_i are greater than the current time. For preemptive scheduling, this also implies that thread T_i should be preempted at the time `min_start`, which is the minimal start time of all $T_j, j < i$. If `min_start` >= the finish time of T_i or i=1, then T_i is not preempted. The calculation of `min_start` can be performed during the search of EL and therefore, does not require additional work.

It should now be apparent why the assignment of threads to slots need not be recorded in the slot list (in contrast to [22]). Namely, the EL is used for thread scheduling, whereas the slot list is used only for schedulability analysis of the threads being scheduled. This leads to another simplification in slot list management. Namely, since the only information of interest is which time intervals are available, slots may be merged when they are adjacent. For instance, slots [2..18] and [18..30] may be merged to slot [2..30]. Merging significantly reduces the number of slots in the list and speeds up schedulability analysis under high loads.

For dynamic scheduling in response to a fork operation, we call a newly arriving thread *schedulable* only if its scheduling does not jeopardize previously scheduled threads. Such a schedulability analysis, then, first places the new thread in order into the EL list, which records all previously scheduled threads, then reschedules a subset of threads in the updated EL using the above slot list based algorithm. If any of the threads in the subset is unschedulable, the new arrival is considered unschedulable. Otherwise, the new arrival is schedulable. The threads in the subset are those threads whose scheduling intervals conflict with the scheduling interval of the new thread. Such subset is easily determined using the slot list [26].

Given the combination of the slot list and the EL, the complexity of the algorithm is $O(n \log n)$ in the worst case (see [26] for detailed analysis). As will be shown later, on average, only a small fraction of total threads must be rescheduled, so that the time to test the schedulability of a new thread is much less than $O(n \log n)$.

Summary of results and extensions. We summarize some interesting results regarding the dynamic algorithm. First, when a system is heavily loaded, relatively more slots in the slot list are likely to be merged. Therefore, the average time for schedulability analysis should be much less than $O(n \log n)$, which is the worst case time.

Second, the algorithm is easily extended to deal with thread precedence. In [4], Blazewicy proposed to enforce precedence constraints among threads by having the deadline of the predecessor thread precede in time the earliest possible start time of the successor thread. This is overly pessimistic since the actual completion time of the predecessor thread may be much earlier than its deadline. This issue does not arise in the algorithm presented here, because a successor thread may be made ready to run as soon as the

predecessor completes its execution by simply giving the successor thread a start time of $S+1$ and a deadline $\geq D+1$, when the predecessor has start time S and deadline D. This ensures that the predecessor thread is positioned before the successor thread in the EL list. Moreover, because the start time of the successor thread is later than the start time of the predecessor thread, the successor thread will not be chosen for execution by the multiplexor until the predecessor thread completes its execution.

The third interesting result is that in practice, the computation time of a thread can only be estimated approximately[17] and pessimistically. If a dynamic algorithm performs scheduling based on such estimates, it would be forced to reclaim unused processor cycles at the time of thread completion. To retain optimality, such reclaimed time slots would have to be made available for new threads and for currently scheduled threads, therefore causing additional overhead. In our algorithm, the reclamation of unused processor cycles is automatically carried out by simply (1) removing a thread from the EL after it completes and (2) executing the dynamic algorithm when testing the next new thread's schedulability.

A first interesting extension of the algorithm is its use for the scheduling of entire thread groups[20], where we assume that there exists a start time *s_set* and a deadline *d_set* for the group. For a group that is considered schedulable only if all of its threads are schedulable, our algorithm can perform the schedulability analysis for all of its members at once, rather than testing members one at a time. If the number of threads in such a group is m, the resulting time is $O((m+k)\log(m+k))$, where k is the number of threads (excluding the threads in the group) involved in the rescheduling.

A second extension of the algorithm concerns periodic threads. Recall that a schedule of n threads is based on threads' deadlines as well as start times, where start times are not necessarily equal to arrival times. As a result, a periodic thread can be represented as a set of sporadic threads with known start times and deadlines. A periodic thread is then said schedulable if and only if all of its periods are schedulable. Note that a more practical approach, however, would first use the rate monotonic algorithm to determine the schedulability of a set of periodic threads in the system and then use our algorithm for some time interval T to perform the detailed schedulability analysis for threads including all the sporadic threads and all the periods of the periodic threads in the interval T. The interval T may be set to be a time span from the present time up to a point beyond which the executions of the sporadic threads that have already arrived will not occur. The length of the interval T is a system parameter that can be tailored by the system programmer, according to the knowledge of the functionality of the system's sporadic threads.

A third extension concerns the consideration of the overhead of thread preemption. If we assume that *SWAP_OUT_TIME* is the time to preempt a thread and *SWAP_IN_TIME* is the time for resuming a thread, then a simple extension of the dynamic algorithm can take into account thread preemption cost. In the extension, both *SWAP_OUT_TIME* and *SWAP_IN_TIME* are added to the execution time of the thread being presently scheduled, whenever the algorithm crosses a slot in searching for available time.

4 Experimental Evaluation

We have developed an analytic model for capturing the algorithm's average case behavior. The model, described in detail in [26], applies the queueing theory to calculate the average number of slots in the slot list. It has shown that the average number of slots in the slot list decrease as the system load increases, which proves our conjecture that the average case complexity of the algorithm is much better than that of the worst case. In this section, we present experimental results that validate the analytic model proposed in [26].

The dynamic scheduling algorithm discussed in this paper is part of a real-time, multiprocessor threads package[27] being used by our group as a basis for the construction of real-time and multiprocessor operating system kernels and applications[8, 7]. To permit its use in hard real-time systems, the real-time threads package guarantees the schedulability of threads dynamically at the time of thread creation (e.g., when

forking threads), provided that threads are the smallest schedulable units in the system and that their deadlines, start times, and maximal computation times are known at their arrival times (e.g., at fork time). In addition, such guarantees are maintained when threads communicate with each other using threads package calls, such as locking shared data, sleeping or waiting on condition variables (see [19] for a complete definition of the real-time threads package).

An implementation of the real-time threads package has been developed on a 32-node GP1000 BBN Butterfly (a 68020-based machine). The BBN Butterfly's memory resides on individual nodes, but any processor can address any memory through the machine's interconnection network. A local memory reference requires approximately 600 nanoseconds and a remote reference requires 4 microseconds, assuming zero switch contention (a reasonable assumption for many real-time applications exhibiting sparse interprocessor communication[21]).

To avoid switch contention, the implementation of threads on the target hardware distributes the system data structures across the nodes of the parallel machine. Furthermore, scheduling bottlenecks are avoided by having each node performs its own thread scheduling by use of a scheduler thread – termed the *local scheduler*[20]. The local scheduler makes decisions regarding all threads assigned to the node, and it is implemented as a special periodic thread resident on each processor. This special periodic thread is different from the other regular periodic threads in that its period (which is a system parameter) changes along with the workload of the corresponding processor.

In addition to possessing a local scheduler, each node also contains a low-level scheduler – termed the multiplexor. It is implemented as a function call as well as an interrupt handler. When a thread completes its computation or when a thread is preempted by a software interrupt, the multiplexor is invoked to choose the next thread to be run[11].

The experiments conducted with the real-time threads package evaluate the actual delay caused by the scheduling algorithm and the number of slots in the slot list under various system loads. The synthetic workloads associated with those experiments use both Weibull and exponential functions as thread inter-arrival time distributions. The Weibull distribution has two parameters: α and γ. The parameter α directly influences the arrival rate:

1. if $\alpha < 1$, the arrival rate is decreasing with time;
2. if $\alpha = 1$, the arrival rate is constant with time, resulting in an exponential distribution; and
3. if $\alpha > 1$, the arrival rate is increasing with time.

The Weibull distribution is chosen since the external events to be handled by dynamically created threads often occur in a dependent, bursty fashion[28]. This is captured more accurately with the Weibull distribution with $\alpha > 1$ than with the exponential distribution.

In the experimentation, a thread's start time is set equal to its arrival time plus a uniformly distributed random number. For the computation time of threads, both the uniform and the exponential functions are used as its distribution. Finally, the deadline of threads is defined as. deadline = start time + computation time + a random slack time, which is also uniformly distributed.

Figure 1 in the appendix depicts the observed overheads in scheduling a new thread as a function of the number of threads already in the system. In these measurements, system workload is 0.8. As with the other experimental results reported below, scheduling cost is almost independent of the distributions of inter-arrival and computation times used in the experimentation. The worst case overhead depicted in Figure 1 is obtained by rescheduling all of the threads in the EL when a new thread is scheduled, using the balanced binary tree in the algorithm. For ease of display, not all points are shown in the figure.

The measurements reported in Figure 1 demonstrate that dynamic thread scheduling is feasible even for a very large number of threads per processor, resulting in scheduling overhead of less than approximately 20 milliseconds for up to 900 threads. Since most devices for robotics (and other real-time) applications require execution rates of no more than 100 Hz, this implies that the highest rate threads in such applications should not be scheduled dynamically, but that even for large-scale, real-time applications, most medium rate threads (threads with average maximum execution times of 100 milliseconds) may be scheduled dynamically with no more than 20% scheduling overhead.

Figure 2 in the appendix illustrates the relationship between two sets of measurements of the dynamic algorithm in a reduced scale, one set using the binary tree, the other set bypassing it. The figure illustrates that the binary tree's benefits are consistently apparent only for a relatively large number of threads (e.g., more than 200), since the balancing operation of the binary tree itself costs significant time. The high variation in scheduling cost for large numbers of threads is due to the fact that a substantial threads in the EL is involved in rescheduling necessitated by thread arrival.

Table 1 shows the experimental data obtained in order to (1) verify the analytic model we have developed and (2) calculate the number of slots in the slot list with different distributions. To achieve exponential inter start times for threads[1] or to achieve Weibull inter-start times, we let the threads' start times equal their arrival times in these experiments. At each load level, measurements are performed using thread sets ranging from size 10 to 600. The table depicts the averages of those measurements. The observed variances are small and are therefore, not reported here.

ρ	$l(n)$			
	Poisson Start Times Uniform Comp. Times	Poisson Start Times Exponential Comp. Times	Weibull Inter-Start Times Uniform Comp. Times	Weibull Inter-Start Times Exponential Comp. Times
0.3	0.73	0.72	0.75	0.74
0.4	0.64	0.64	0.65	0.64
0.5	0.57	0.53	0.55	0.54
0.6	0.46	0.44	0.45	0.42
0.7	0.42	0.30	0.35	0.32
0.8	0.33	0.21	0.23	0.24
0.9	0.21	0.12	0.09	0.08
0.95	0.16	0.05	0.05	0.05

Table 1: Experimental Value of $l(n)$ with Different Workloads ρ

5 Conclusions

This paper develops and evaluates a dynamic preemptive scheduling algorithms for hard real-time systems. The results presented demonstrate that the dynamic algorithm may be used for on-line scheduling of a large number of threads (up to 2000 in our current experimentation) per processor without undue performance penalties. The threads being scheduled include sporadic and periodic threads, and straightforward algorithm extensions can handle thread groups and threads related by precedence constraints.

Not shown in this paper are our additional experiences with a real-time threads package that concern dynamic scheduling for thread synchronization, such as scheduling performed when threads make dynamic decisions to wait[21, 27] on certain conditions (with well-defined timeout values). Such scheduling is performed by inspection of condition waiting queues and thread start times and deadlines.

This paper has confined itself to presentation of uniprocessor real-time scheduling. Our current work concerns the extension to multiprocessor scheduling, based on the uniprocessor algorithms presented here and on our previous work in multiprocessor, real-time scheduling[27, 20, 22].

[1] Exponential inter-start time is equal to Poisson start time.

A Appendix

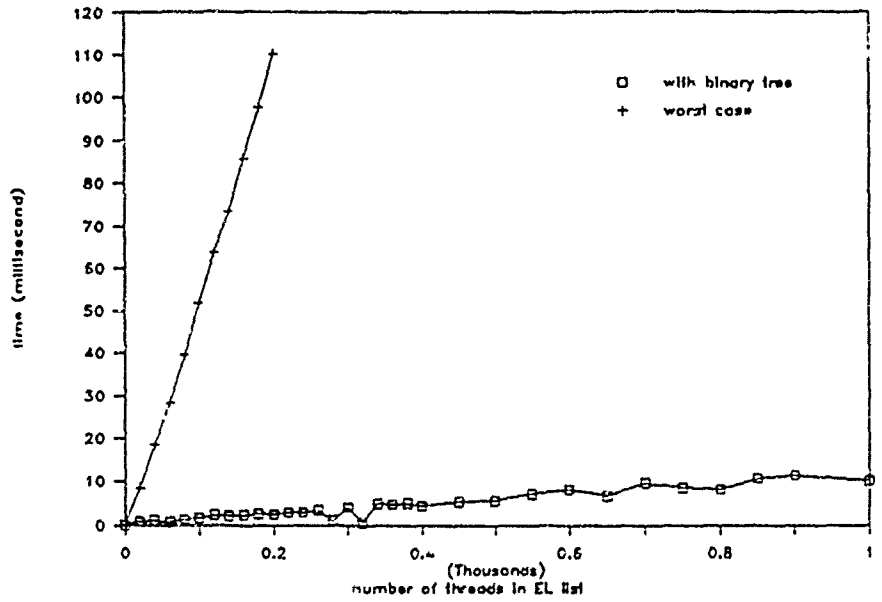


Figure 1: Observed Overhead in Thread Scheduling

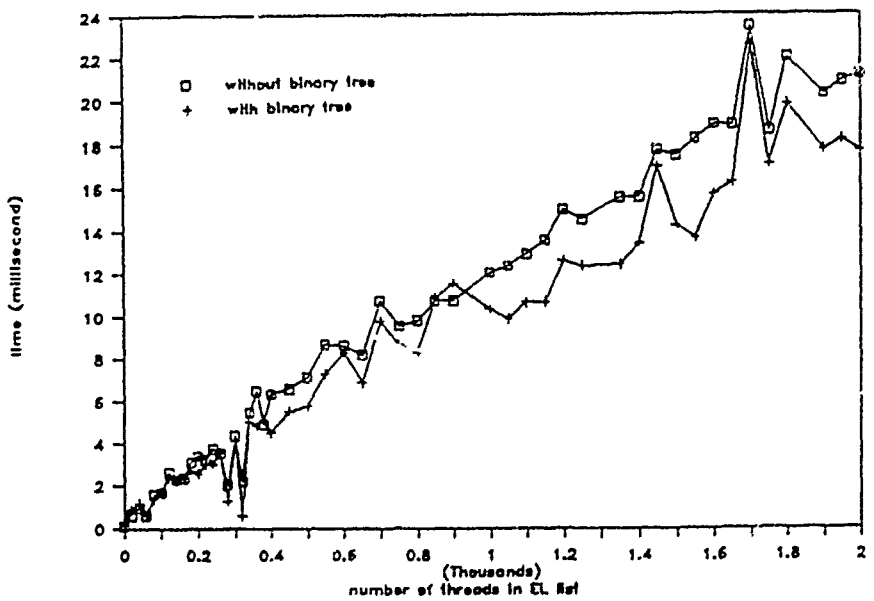


Figure 2: Scheduling Overhead of the Dynamic Algorithm

References

[1] T. Bihari, D. Pugh, T. Walliser, and E. Ribble. Timing analysis of a robot motion-planning algorithm. In *Seventh IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 104–107, May 1990.

[2] T. Bihari and K. Schwan. A comparison of four adaptation algorithms for increasing the reliability of real-time software. In *Ninth Real-Time Systems Symposium, Huntsville, AL*, Dec. 1988. Also submitted to *Real-Time Systems.*

[3] T. Bihari and K. Schwan. Dynamic adaptation of real-time software for reliable performance. Technical report, Department of Computer and Information Science, The Ohio State University, OSU-CISRC-5/88-TR, May 1988. To appear in ACM Transactions on Computer Systems.

[4] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Modelling and Performance Evaluation of Computer Systems.* North-Holland, 1976.

[5] Gene D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27(9):926–936, Sept. 1984.

[6] Michael L. Dertouzos. Control robotics: the procedural control of physical processes. In *Proc. of the IFIP Congress*, 1974.

[7] Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing, Chicago, ILL*, pages 462–469, June 1989.

[8] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. Technical report, GIT-ICS-90/06, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, Jan. 1990. Submitted for publication.

[9] W. A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[10] Lui Sha John P. Lehoczky and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of Real-Time Systems Symposium, San Jose, CA*, pages 261–270. IEEE, 1987.

[11] A.K. Jones, R.J. Chansler, I. Durham, J. Mohan, K. Schwan, and S. Vegdahl. Staros, a multiprocessor operating system. In *Proceedings of the 7th Symposium on Operating System Principles, Asilomar, CA*, pages 117–127. Assoc. Comput. Mach., Dec.10-12 1979.

[12] Korein, Maier, Taylor, and Durfee. A configurable system for automation programming and control. In *IEEE International Conference on Robotics and Automation, San Francisco, CA*, pages 1871–1877. IEEE, April 1986.

[13] Jeff Kramer and Jeff MaGee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[15] Robert B. McGhee. *Vehicular Legged Locomotion*, pages 259–284. Jai Press Ltd., 1985.

[16] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[17] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[18] Lui Sha Ragunathan Rajkumar and John P. Lehoczky. On countering the effects of cycle-stealing in a hard real-time environment. In *Proceedings of Real-Time Systems Symposium, San Jose, CA*, pages 2–11. IEEE, 1987.

[19] K. Schwan, A. Gheith, and H. Zhou. Chaos-arc: A kernel for predictable programs in dynamic real-time systems. In *Seventh IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 11–19, May 1990.

[20] Karsten Schwan, Thomas E. Bihari, and Ben Blake. Adaptive, reliable software for distributed and parallel, real-time systems. In *Sixth Symposium on Reliability in Distributed Software, Williamsburg, Virginia*, pages 32–44. IEEE, March 1987.

[21] Karsten Schwan Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[22] Karsten Schwan and Ben Blake. A fast scheduling mechanism for real-time systems. Technical report, Computer and Information Science, The Ohio State University, OSU-CISRC-5/87-TR16, Sept. 1987. Being revised for publication.

[23] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. From chaos-min to chaos-arc: A family of real-time kernels. To appear in 1990 Real-Time Systems Symposium., May 1990.

[24] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[25] Karsten Schwan and Rajiv Ramnath. Adaptable operating software for manufacturing systems and robots: A computer science research agenda. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, pages 255–262. IEEE, Dec. 1984.

[26] Karsten Schwan and Hongyi Zhou. Optimum preemptive scheduling for hard real-time systems: Toward real-time threads. Technical report, College of Computing, Georgia Institute of Technology, GIT-ICS-90/28, Sept. 1990.

[27] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. A multiprocessor real-time threads package. Technical report, College of Computing, Georgia Institute of Technology, Oct. 1990. In preparation.

[28] D. Siewiorek and Robert S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

[29] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

A Reliable Multicast Protocol for Distributed Real-Time Systems

H. Kopetz, G. Grünsteidl

Institut für Technische Informatik
Technical University Vienna
A-1040 Vienna, Austria

Abstract

Distributed computer architectures are well accepted in the domain of real-time applications. To realise fault-tolerance, node computers providing the same service can be clustered into Fault-Tolerant Units (FTUs). Each FTU provides a specified service as long as at least one of its node computers is operational. The communication between these FTUs has to be reliable and timely, i.e. there must be a tight upper bound on the time it takes to send a message from one FTU to the other FTUs. This paper presents a communication system suitable for real-time applications that meets these requirements.

1 Introduction

A computer system for real-time applications must respond to a stimulus from the controlled object within an interval dictated by the environment, called the response time. This response time must be guaranteed under all specified load and fault conditions. A common technique to realize fault tolerance in distributed real-time systems is the active replication of node computers in order to provide the specified service despite of failure of a node computer. Therefore, the set of node computers is partitioned into disjoint subsets, which forms Fault-Tolerant Units (FTUs). All node computers belonging to a FTU provide the same service. Each FTU provides a specified services as long as at least one node of it is operational.

The communication between the FTUs of the distributed real-time computer architecture has to be reliable and timely despite of communication failures or node failures (i.e. there must be a tight upper bound on the time it takes to send a message from one node computer to the other node computers) [Kur84].

To provide the same service all node computers of a FTU have to receive the same messages. Therefore the communication system should provide a one-to-many communication (multicast) between the node computers. This multicast service is also useful at the application level.

Protocols providing broadcast (multicast) communication between the node computers of a distributed system are known for asynchronous architectures [Cha84,Pet89,MS90] and synchronous architectures [Bab85,Cri90]. In asynchronous architectures there exists no bound on message delays and no global time base (synchronized local clocks) [Cri91].

To realize communication under real-time constraints our protocol is based on a synchronous architecture. Contrary to known synchronous protocols that realize communication on the level of node computers, we take strongly into account the clustering of the node computers into FTUs.

In this paper we present a synchronous communication system to realize reliable multicast communication between the FTUs of a distributed system under real-time constraints. In the next two sections the system architecture and the objectives of the protocol are introduced. Section 4 (protocol description) describes how the communication, the shut-down and switch-on behavior of node computers, and the management of the membership information is realized.

2 System Architecture

We assume that a distributed system for real-time applications consists of a set of n autonomous selfchecking node computers which are interconnected by a broadcast channel (local area network (LAN)). Each selfchecking node is a selfcontained computer with a CPU, a local memory, an interface to the LAN consisting of an incoming link and an outgoing link, and a local real-time clock. It has to contain error detection mechanisms in hardware or software to detect errors within itself and turn itself off locally. Communication among the nodes is achieved by the exchange of broadcast messages only, i.e. a message sent by any one node can be received by all other nodes. All clocks of correctly functioning nodes are synchronized with a known constant maximum deviation Δ [Kop87], i.e. at any point in time the deviation between any two such clocks is always smaller than Δ, thus establishing a global time base of known precision.

In order to tolerate failures of the broadcast channel we propose that the nodes are connected by two actively redundant channels. The nodes have access to the channels in a strict deterministic sequential order by using a suitable access strategy, e.g. a synchronous time division access (TDMA) strategy. The access sequence is identical for both redundant communication channels. Each node has exclusive access for a constant period called slot. Given n nodes and beginning with node 1 it takes n slots for every node to communicate with every other node in the system by sending a broadcast message. This length of time is called a round.

Each node of a distributed real-time system has to provide a specified service determined by its application software (real-time tasks). To tolerate the failure of a node two nodes operate in active redundancy. The actively redundant nodes which provide the same service form a fault tolerant unit (FTU), i.e. all nodes of a FTU which are operating at a particular point in time contain the same inner state at about the same time, determined by the synchronization accuracy Δ introduced above. As long as at least one node of the FTUs is operational it will deliver the specified service. We assume that the nodes forming a FTU have sequential sending slots assigned to them. The duration of all sending slots assigned to a FTU is called a FTU slot.

To increase the degree of redundancy we use the concept of shadow nodes [Kop90]. A shadow node operates in redundancy with two other actively redundant nodes of a FTU. It receives and processes messages, and therefore maintains an internal state equivalent to the states of the active nodes. The difference from the other nodes is that no sending slot is assigned to it. Consequently, it does not broadcast any message on the broadcast channels. Thus the

use of shadow nodes does not affect the timing behavior of the communication between the nodes. In case of a failure of an active node, the slot of the faulty node will be taken up by the shadow node.

3 Objectives of the Protocol

The objectives of the protocol are to realize:

- reliable and timely multicast communication between the FTUs of a distributed system
- correct shut-down behavior of faulty nodes
- correct switch-on behavior of shadow nodes
- consistent and timely detection of node failures
- consistent and timely detection of node restarts

The shut-down behavior of nodes and the switch-on behavior of the shadow node inside a FTU have to be consistent to guarantee a correct FTU behavior.

4 Protocol Description

4.1 Failure Hypothesis

Failures causing the loss of one or more messages are considered. These failures may concern the nodes or the communication system (links and channels). We assume that there is enough redundancy in each message (e.g. a signature and/or CRC check bits) that a mutilation of the message contents can be detected. The receiver node discards such messages.

Nodes can suffer crash failures and messages can get lost because of omission or crash failures of the channels or links.

4.2 Communication

We assume that each non-faulty node will send a broadcast message in each round in its assigned slot at a predefined point in time known a priori to all other nodes of the system. In case no application data have to be transmitted, the node will still broadcast an empty message as a life sign. (It is a property of the TDMA strategy that empty slots cannot be used by other nodes anyway.)

A non-faulty node sends exactly those messages that are prescribed by its task specification. A node is active at its sending point if it broadcasts a message at its assigned slot time. A copy of this message is sent simultaneously over each of the two redundant channels by a non-faulty node. Every message contains in the header a protocol field providing information about the messages the sending node has received or not received in the last n slots. This information corresponds to positive and negative acknowledgments and therefore will be called the ACK-field of the header. The ACK-field consists of $2n$ bits, corresponding to the $2n$ messages which are exchanged in one round.

In normal operation every node will receive all four redundant messages — two via the two redundant channels per node and two in the two consecutive time slots comprising the FTU slot. At least one out of these four redundant messages from a non-faulty FTU must be received by a node to continue the service. These four messages contain identical application data. The ACK-field of messages sent from different active nodes of a FTU can differ.

4.3 Shut-Down and Switch-On Behavior of Nodes

To generate a correct message a node has to receive messages from the other nodes. A message is correct:

- if it is sent at the specified time known a priori to all other nodes
- if it contains application data as intended
- if its ACK-field is compatible with the ACK-field of the majority of the other messages (i.e. the sending node has not missed application messages that have been received by the majority of the other nodes)

In case a node detects that it cannot generate a correct message it has to shut-down. This decision depends also on the application and on the proper functioning of the node.

At the communication level we can decide whether a node is receiving and sending the specified messages. A failure to do so is an additional reason for node shut-down. This cannot always be detected by the sending node, but sometimes only by the receiving nodes. They refuse to accept and to acknowledge messages positively with an incompatible ACK-field.

Message Rejection Property: If a node s receives the first "incorrect message" from another node r, then it will discard this message and all further messages of this node, until the termination of a join protocol (of the node r).

We define the shut-down behavior on the communication level. After a node has sent out a message it waits for positive and negative acknowledgments for this message. This information is piggypacked in the messages sent in the next $n-1$ slots. To tolerate the loss of (acknowledgment) messages we specify the shut-down behavior and switch-on behavior of nodes as follows:

Node Shut-Down Property: A node has to shut-down itself (terminate its service)

- if it cannot generate a correct message
- if it has not received a positive acknowledgment from more than half of the active nodes for its last output message

Node Switch-On Property: A shadow node has to switch-on itself, if the last output message of an active node within its FTU has been negatively acknowledged by more than half of the other active nodes.

In order to evaluate the conditions for shut-down and switch-on of nodes, a node requires a timely knowledge about those nodes from which it can expect a (negative or positive) acknowledgment. Therefore, each node of the distributed system has to have knowledge about the operational state of the other nodes (membership information) [Cri88,Kop89b].

Based on the fact that all acknowledged messages are correct messages and that non-faulty nodes have to send messages periodically, the messages could be used as life sign messages. If a node received a correct message sent by another node, it can conclude the activity of this node directly. In its next message this node will send a positive acknowledgment (in the ACK-field) for this message. Despite the fact that no message forwarding is realized explicitly, the used acknowledgment schema corresponds to a forwarding technique for the life sign information. In the case a node has not received a message from a node directly, it can be informed by the other nodes about the activity of the sending node which is in question.

To detect intermittent or permanent output link or incoming link failures of nodes or channel failures which do not cause the shut-down of nodes a passive monitor node (which can be in addition to the n nodes) must observe the correct operation of all nodes and initiate maintenance activities if required.

5 Protocol Properties

Error Detection: This multicast protocol provides consistent error detection at the sender and at the $n-1$ receivers of every information exchange. Error detection at the receivers is of particular importance in a real-time environment.

Fault Tolerance: The protocol can tolerate up to three message failures in each FTU slot. If in a single round all four messages fail at any one node, the protocol still guarantees consistent behavior of the set of nodes.

Promptness: In order to guarantee consistent behavior, the protocol takes one round of information exchange. Since one round of information exchange is the theoretical minimum in order to guarantee consistent behavior in the face of a total FTU loss, this protocol provides optimal promptness.

6 Conclusion

We have presented a synchronous communication system suitable for real-time applications. The described protocol realizes reliable multicast communication between the Fault-Tolerant Units (FTUs) of a distributed real-time system. It is optimal from the point of view of promptness, provides error detection at the sender and receiver and is fault tolerant.

It is planned to implement this protocol in our new version of the MARS system [Kop89a] on a dedicated hardware processing unit, such that an application is not impacted by resources required for the protocol execution.

It depends on the desired degree of fault-tolerance how many redundant channels are used and how many nodes form a FTU. However, in order to simplify the description how the communication is realized and to describe the main properties of the presented protocol it was sufficient to explain the system with two active communication channels and two active nodes per FTU.

References

[Bab85] Ö. Babaoğlu and R. Drummond. Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts. *IEEE Transactions on Software Engineering*, SE-11(6):546–554, June 1985.

[Cha84] J. M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug. 1984.

[Cri88] F. Cristian. Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System. In *Proc. 18th Int. Symposium on Fault-Tolerant Computing*, pages 206–211, Tokyo, Japan, June 1988.

[Cri90] F. Cristian. Synchronous Atomic Broadcast for Redundant Broadcast Channels. *Real-Time Systems*, 2(3):195–212, Sept. 1990. Kluwer Academic Publisher.

[Cri91] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[Kop87] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, 36(8):933–940, Aug. 1987.

[Kop89a] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[Kop89b] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. In *Int. Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, CA, USA, Aug. 1989.

[Kop90] H. Kopetz, H. Kantz, G. Grünsteidl, P. Puschner, and J. Reisinger. Tolerating Transient Faults in MARS. In *Proc. 20th Int. Symposium on Fault-Tolerant Computing*, pages 466–473, Newcastle upon Tyne, UK, June 1990.

[Kur84] J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-Access Protocols and Time-Constrained Communication. *ACM Computing Surveys*, 16(1):43–70, March 1984.

[MS90] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, Jan. 1990.

[Pet89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug. 1989.

GARTEN: A Programming Environment for Real-time Software Development

KEITH J. RANSOM & CHRIS D. MARLIN
Department of Computer Science,
The University of Adelaide,
Adelaide, South Australia 5001

keith@cs.adelaide.edu.au
marlin@cs.adelaide.edu.au
Fax: +61 8 223 1206

WEI ZHAO
Department of Computer Science,
Texas A & M University,
College Station
Texas 77843-3112

zhao@neuron.tamu.edu
Fax: +1 409 847 8578

1 Introduction

During the past decade, much attention has been focused on the development of hard real time systems. Real-time systems now play a crucial role in many real world applications including missile-control, robotic equipment, the space shuttle, and so on

Stream-lining the software development cycle for real-time applications is becoming increasingly important Indeed, with the advent of more and more ambitious applications such as undersea exploration robots, and NASA's space station project, any small percentage saving in the software development cycle may result in a substantial reduction in overall cost

One can expect that the construction of real-time software systems will be facilitated if the programming language and development environment used have been designed to support the particular approach to real-time systems construction which is being employed. Many general purpose programming languages which purport to be of use in programming real-time applications often prove to be less than desirable when used in practice, as they most often provide, at best, limited control over scheduling decisions and policies At the same time, the real-time systems programmer does not wish to sacrifice the benefits of modern high-level structured programming languages (by using assembly language, for example) simply in order to gain more control over the real-time aspects of a system.

In any form of software development, and to a larger extent in real-time systems development, the testing-debugging phase constitutes a large proportion of the total development time. Due to the critical role of many real-time systems, particularly those where malfunction may be life-threatening, it is often necessary to enforce the highest standards of quality assurance to be employed during the testing phase. As a result, this phase of real-time software development is often an order of magnitude larger than all others. Thus, there is much to be gained from reducing the complexity of the testing phase whilst maintaining the same guarantees of system performance.

The real-time programming language GARTL and its associated development environment GARTEN introduce features to both reduce the time spent in the programming-testing cycle and increase the efficiency of the software systems produced. The language GARTL [12, 14] has been designed as an extension to the C programming language in order to provide a sound base with which many programmers will already be familiar.

One of the principal problems contributing to the high cost of programming-testing cycle of hard real-time systems is that (by definition) the various hard real-time deadlines imposed on the system must be met without fail. Each of the various components or *tasks* of the system must be tested to ensure that any corresponding time constraints are met. Thus, there may be a long cycle of activity whereby a task is coded, tested and then re-coded, until it eventually executes within its allotted time constraints. GARTL substantially reduces the amount of re-coding required by allowing the programmer to take advantage of the concept of imprecise computation [2, 5, 6, 7, 8, 9, 13, 15, 16]. Imprecise computation simply means that the accuracy of a value computed by a task, or the amount of work performed by the task, is in some way proportional to the amount of time the task is given to execute. GARTL supports imprecise computation by allowing the specification of multiple task versions, each having different resource requirements, most importantly processing time. Thus, the static process of modifying code and re-testing, is superseded by the ability of the GARTL run-time system (called GARTOS) to dynamically choose a task that will meet the appropriate time and resource constraints.

The main intent of this paper is to describe the GARTEN environment, focussing primarily on the GARTL language and GARTAAN, an analyser-debugger for GARTL. Section 2 provides an overview of GARTEN and briefly describes the major constituents. Section 3

provides a brief introduction to the GARTL language and highlights the main features of interest. The analyser-debugger is described in Section 4.

2 GARTEN

As the complexity of new real-time applications continues to increase, so to does the need for an improvement in real-time software development technology. Interest in recent years in the field of software engineering has seen the emergence of many ideas for integrated programming environments, and much of this work can be applied to real-time applications programming. However, the very nature of real-time applications, reflected by some of the problems highlighted in the previous section, is such that an environment dedicated to developing these applications must contain many facilities in addition to those present in other programming environments.

We describe the nature of GARTEN, a complete development environment for real-time applications which supports the GARTL programming language. A salient feature of each of the tools to be provided by GARTEN is that they must necessarily be oriented to the timing characteristics of real-time software.

The environment is comprised of four major components, namely

- a language specific editor for GARTL source code,
- the GARTL language compiler,
- GARTOS, the run-time system for GARTL applications, and
- GARTAAN, an application tester-debugger, supporting database management and browsing facilities.

A language-specific editor [10] along the lines of the multiple window, multiple view editing facilities of the MultiView programming environment [1, 11] is envisioned. This would allow a more structured approach to the creation of GARTL source code. Such an editor would also relieve the compiler of the traditional parsing stage, as it would simply operate on the abstract syntax tree constructed by the editor. The current version of the GARTL compiler is merely a GARTL to C translator, working with GARTL source text. However, it is a simple matter to modify the compiler to instead manipulate an abstract syntax tree.

Since GARTOS is not the focus of this paper, it will not be described in detail here; however, there are some points worthy of note. The GARTOS operating system is designed to be layered upon a host operating system, providing a suitable level of abstraction to enable GARTL applications to execute over a heterogeneous network of target architectures. In addition, a high level of support is provided in the form of operating system calls used by the C code generated by the GARTL compiler. Perhaps the most important aspect of GARTOS is its ability to predict the computation time of a task (tasks are introduced in Section 3 below). To do so, GARTOS makes extensive use of a database of execution information specific to the given application. The nature of the database and information stored within it will be described in more detail in Section 4, along with the application tester-debugger GARTAAN.

3 The GARTL language

As mentioned, the GARTL language has been designed as an extension of the C programming language [4]. This section serves simply to highlight those extensions representing the most important aspects of GARTL. More detailed descriptions and examples of GARTL programming can be found in [12, 14].

3.1 Support for imprecise computation

As mentioned previously, GARTL supports imprecise computation by allowing the programmer to specify multiple versions of a task, each with different resource requirements (most notably, requiring differing amounts of *processor time*). The run-time system will always choose the most desirable version of a task to run, with the constraint that all of its resource requirements must be satisfied. A more desirable version of a task is naturally one that provides a more precise result. Typically, the more precise a result, the longer a task will take to execute. The interested reader may refer to [12] where the concept of multi-version tasks are discussed further.

A similar method for supporting imprecise computation is provided by the programming language FLEX [5, 6, 13]. In FLEX, there is a mechanism whereby partial (or imprecise) results of a task are maintained. If the task must be pre-empted (due to the expiration of its deadline, for instance), these partial results are made available to the caller of the task. However, the approach used in FLEX requires a more restrictive form of task definition, in that a result must always be ready in case the task is aborted. In GARTL, the system scheduler is called whenever a task is invoked and will decide whether a given task version will meet its deadline or not. If the scheduler decides that a task will meet its deadline, then and only then will it be executed. Once such a decision has been made the task is *guaranteed* to meet its deadline. For this reason, a particular task version need not maintain partial results, since if it is executed then it should execute until completion.

3.2 Scheduler interaction

Apart from the main task of a GARTL program, the execution of every subsequent task is invoked by another

task. A new (or *child*) task is invoked by an existing (or *parent*) task by means of a scheduling request. With such a request, the parent task sends the system scheduler the following information:

- the parameters to pass the child task,
- which versions of a multi-version task are acceptable,
- the manner in which the child task should be executed, and
- whether the results of the scheduler's decision are required.

There are essentially two ways in which a child task may be executed, namely *synchronously* and *asynchronously*. In addition, there are two ways in which the scheduler may be called, again *synchronously* and *asynchronously*. Thus, there are four possible ways of scheduling a given task.

Often, it is desirable to know whether or not a particular task was successfully scheduled. In order to obtain such information, it is possible to indicate to the scheduler that the results of the request are of importance and should be returned to the caller.

The interaction that is permitted by GARTL applications and the system scheduler is an important and innovative feature of GARTL. Whilst many other programming languages allow dynamic task invocation, such languages provide no means of obtaining feedback from the system to the invoking task concerning the timing of the invoked task. The invoking task knows only that the execution of the invoked task will be logically correct. No information concerning the invoked task's start time or ability to meet its deadline are provided. Thus, missed deadlines are hard to detect in the invoking block, making appropriate recovery actions difficult to take.

3.3 Monitored variables

Upon receipt of a scheduling request for a given task, the scheduler will look at the values of a subset of parameters to be passed to the invoked task, and uses its knowledge gained from past executions of the task to determine whether it is possible to successfully schedule an acceptable version of the task. The parameters to be regarded as significant in predicting execution times from past executions of a task are declared with the task as *monitored variables*.

Within a task definition, monitored variables are used just as any ordinary variable. It is only the initial values to be passed to a scheduled task that are important when predicting the execution time for the given task. If a task has no monitored variables, it is assumed that the execution time for that task is independent of the task's input values. Naturally, the onus is on programmer to ensure that monitored variables are chosen wisely.

The declaration of a monitored variable constitutes a partitioning of the possible values which the corresponding parameter may legally have. Thus, it is a list of collections, each enclosed in square brackets. A single collection groups together those values that are easily derivable from one another.

3.4 Finite execution times

An issue of fundamental importance in GARTL programming is that of predictability of execution times. Since GARTL has been designed primarily with hard real-time systems programming in mind, it is vital that the GARTOS system scheduler be able to predict the execution time of a task, in order to provide guarantees concerning its execution. This means that given a set of parameter values to be passed to a task, the scheduler must be able to decide, before it is successfully scheduled, whether or not the task will finish on time. For this reason, it is necessary to ensure that the execution times of all GARTL tasks are bounded.

In order to ensure bounded task execution times, a number of language constructs are provided in GARTL that place restrictions on otherwise unbounded constructs. These restrictions include mandatory iteration or time limits for all loop constructs, and recursive invocation limits for GARTL functions. In addition, there is no asymmetric control transfer statement in GARTL, relinquishing the *goto* statement provided by the C language.

4 GARTAAN

Perhaps the most innovative feature of the GARTL language is the ability to take full advantage of the imprecise computation mechanism. By providing multi-version tasks, GARTL allows for as sophisticated a model of imprecise computation as the programmer might care to devise. However, with such a language facility comes the need to accurately predict the execution times for each version of a given task. Hence, the concept of monitored variables is supported by the language as a means of allowing the programmer to specify which of the input parameters of a task are important in determining its execution time for each of its given versions.

As a further consequence of the GARTL facilities mentioned, the scheduling of a task in a GARTL application becomes quite complex indeed. Thus, each GARTL application maintains a database of execution times to be used for future scheduling predictions. For each task in a given application, an entry in the database is kept, recording an array of execution time information indexed by the version number of the task and the corresponding partition of the range of values for each of the monitored variables. Hence, a database entry actually contains an

$(n+1)$ dimension array for a task with n monitored variables

Before GARTL can be allowed to execute in its *real-world* environment, the entire database must be filled with preliminary values in each of the data cells. Each time a GARTL task is executed, the values in the corresponding data cell will be updated. Thus, more accurate results are continuously evolving whenever an application is executing. At this point the problem arises as to how the database should be filled initially before an application is ever executed. One solution is to execute all the tasks in the application with all possible combinations of monitored variables and version numbers. Such a solution might seem paradoxical, since the same problem will be faced when trying to perform the test executions. Thus, the need arises for an application analyser to provide an environment where the tasks of an application may be executed initially without any previous timing values being present in the database. To this end, GARTAAN has been designed. The functionality and user interface of GARTAAN will be described in Section 4.2.

Ideally a suitable tool for the above purposes would also provide the whole range of features of a traditional debugging tool, such as source-level debugging and other facilities to assist in assuring the logical correctness of a program. However, while our overall aim is to provide an integrated set of both logical-testing and timing-analysis facilities, the remainder of this paper will concentrate on the timing analysis aspect of such a tool

4.1 Design issues

When attempting to schedule a task within the testing environment, it is still necessary to have some estimate of the length of time that the task will take to execute for the given set of input values. Problems arise if the data cell corresponding to the requested execution is empty. In such a case, an estimation function should be provided which returns an estimate of the time given the appropriate version number and monitored variable values. In order to avoid this problem, it is possible to perform the test execution of tasks in such an order that no task is ever scheduled before the corresponding database cell contains the appropriate information. For this reason, the GARTL compiler produces a table representing the call-graph, for the compiled application. From such a call-graph, GARTAAN can determine which tasks are *sinks* of the graph, and hence may be executed without any timing information being available.

Due to the nature of many hard real-time applications, it would not be feasible to perform analysis of tasks whilst executing them in the environment in which they are destined to run. For this reason, it is likely that the programmer will want to substitute dummy routines for certain control routines: these dummy routines may be safely executed within the sheltered environment of the analyser. Typically, such dummy routines will consist of a delay for some period of time, the duration of which may be derived from the values of the monitored variables.

To fill in a data cell for a given version of a task with a particular combination of monitored variables, it will normally be necessary to execute the task some number of times, supplying input values drawn from the respective partitions specified for the monitored variables. The most accurate way of filling in the data cell will involve performing tests using every combination of values in each of the respective partition, and combining the results from each measurement in some suitable manner. This method is, of course, prohibitive and takes no account of the way in which the partitions were constructed in the first place. Rather than perform exhaustive testing, we propose that a sampling policy be adopted, where the frequency of the sampling may be dictated by the user. In each single data cell, we propose spline interpolation as the means of providing timing predictions for value combinations that have not been tested.

Due to the empirical approach used to obtain the timing values with which to fill in an application's database, GARTAAN is constrained with respect to the architecture upon which it may run. While the analyser does not have to run on exactly the same machine that will execute the real application, it does have to execute on the same configuration of machines.

4.2 GARTAAN interface and functionality

For the purposes of this paper, the GARTAAN user interface will not be described in terms of any one particular implementation or target machine; instead (in keeping with the design), it will be described in terms of a generic set of interface capabilities, such as that which may be found on most modern work-stations.

The GARTAAN display is conceptually divided into three main areas, namely

- a control panel, providing access to the frequently used functions of the system,
- a call-graph window displaying the call-graph of the application being analysed, and
- a console window, used to display and accept various textual information

In addition to these display elements, GARTAAN provides a menu facility, by means of hierarchical popup-menus or the traditional menu-bar

As the name suggests, the principal function of the call-graph window is to display the call graph of a given GARTL application. Within this window, the user can scroll around the graph by using scroll-bars attached to the sides of the window. Many functions that may be chosen from the control panel require the user to perform

manipulations within the window, such as framing an area of the graph for example.

As mentioned, the control panel is provided to allow easy access to the most common functions performed in GARTAAN, which are otherwise accessible via the menu facility

The most important function of the analyser is *time*. The whole purpose of analysing a GARTL application (apart from traditional debugging activities) is so that the application timing database may be filled in ready for use in the real-world environment. To *time* a GARTL task means to fill in every cell in the application's database corresponding to the task. Thus, each task will be executed many times using the analyser. After timing operations have been performed, it is possible to view the contents of the database via the *examine* function

In many complex real-time applications, it is not unusual to have a number of tasks, in the order of hundreds. In such cases, the application's call-graph may be extremely large and difficult to manipulate. Consequently, GARTAAN supports a graph truncation function, where a large sub-graph may be treated as a single node for the purposes of viewing the graph.

Previously it was mentioned that there is often a need to replace certain critical routines with dummy routines that may be safely executed in the testing environment. For this reason, the *replace* function is provided. This function allows the user to alternate between the real and dummy version of a particular routine.

References

[1] R. A. Altmann, A. N. Hawke and C. D. Marlin, "An Integrated Programming Environment Based on Multiple Concurrent Views", *Australian Computer J.*, Vol. 20, No. 2, (May 1988), pp. 65–72.

[2] E. K. P. Chong and W. Zhao, "Task Scheduling for Imprecise Computer Systems with User Controlled Optimization", in *Computing and Information*, R. Janicki and W. W. Koczkodaj (Ed.), pp. 141–146 (North-Holland, Amsterdam, 1989).

[3] E. K. P. Chong and W. Zhao, "Performance evaluation of scheduling algorithms for imprecise computer systems", to appear in *Journal of Systems and Software*, 1991.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice Hall Inc., Englewood Cliffs, New Jersey, 1978).

[5] K. Lin, S. Natarajan and J. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems" *Proc. I.E.E.E. Real-Time Systems Symposium* 1987, pp. 210–217.

[6] K. Lin, S. Natarajan and J. Liu "Expressing and Maintaining Timing Constraints in FLEX" *Proc. I.E.E.E. Real-Time Systems Symposium* 1988, pp. 96–105.

[7] J. Liu, K. Lin and C. Liu, "Concord Prototype System and Real-Time Scheduling", *Proc. I.E.E.E. Fourth Workshop on Real-Time Operating Systems* 1987, pp. 27–30.

[8] J. Liu, K. Lin and S. Natarajan "Scheduling Real-Time, Periodic Jobs Using Imprecise Results" *Proc. I.E.E.E. Real-Time Systems Symposium*, 1987, pp. 252–260.

[9] J. W. S. Liu, K.-J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations", to appear in *I.E.E.E. Computer*, 1991.

[10] C. D. Marlin, "Language-specific Editors for Block-structured Programming Languages", *Australian Computer J.* Vol. 18, No. 2, (May 1986), pp. 46–54.

[11] C. D. Marlin, "A Distributed Implementation of a Multiple View Integrated Software Development Environment", *Proc. 5th. Conf. on Knowledge-Based Software Assistance* 1990, pp. 388–402.

[12] C. D. Marlin, W. Zhao, G. Doherty and A. Bohlmann, "GARTL: A Real-time Programming Language Based on Mulit-version Computation", *Proc. I.E.E.E. Int. Conf. on Computer Languages* 1990, pp. 107–115.

[13] S. Natarajan and K. J. Lin "FLEX: Towards Flexible Real-Time Programs" *Proc. I.E.E.E. Int. Conf. on Computer Languages* 1988, pp. 272–279.

[14] K. J. Ransom, *The GARTL Reference Manual* (Dept. Computer Science, Adelaide, South Australia, 1991)

[15] J. Stankovic, "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems", *I.E.E.E. Computer*, vol. 21, No. 10, October 1988, pp. 10–19

[16] W. Zhao and E. K. P. Chong "Performance Evaluation of Scheduling Algorithms for Dynamic Imprecise Soft Real-Time Computer Systems", *Australian Computer Science Communications*, vol. 11, No. 1, February 1989, pp. 329–340.

Schedulability, Program Transformations and Real-Time Programming

Alexander D. Stoyenko *
Thomas J. Marlowe †

January 14, 1991

1 Introduction

Schedulability analysis [8,14,15] and related forms of analyses [16,6,5,11,12,9,10,13,17] provide compile-time verification of deadline satisfaction in a wide spectrum of hard real-time applications. However, the cost of finding, through the analysis, accurate solutions to this generally NP-complete problem adds significantly to the cost of program compilation. In this context, additional polynomial-time static analyses of program semantics, and semantics-preserving polynomial-time program transformations, can be undertaken without significantly increasing the cost of compilation, and may actually result in reduction of total time for analysis, if this analysis or program transformations can reduce the cost of the schedulability analysis.

In this paper, we introduce additional static semantic analysis and transformations used in a limited language to produce simple, analyzable program forms. These forms facilitate accurate polynomial-time schedulability analysis techniques thus decreasing the cost of the analysis, improve the possibility of deadline satisfaction, and in some cases provide a completely static schedule for a moderately complicated real-time program. We anticipate ongoing research to extend these results by eliminating assumptions on the language and the environment, to produce a set of polynomial-time analysis techniques, and to integrate the static semantic analysis and transformations and these techniques in a prototype.

2 Model and Assumptions

The Semantics of Real-Time Programs: The semantics of real-time programs must include deadline satisfaction. We will say that a program transformation is *deadline-isomorphic* if the transformed program will meet deadlines if and only if the original program did; *deadline-preserving* if the transformed program will meet deadlines if the original program did; and *deadline-extending* if the original program would have met deadlines whenever the transformed program does. Finally, a transformation is *deadline-destroying* if it satisfies none of these properties.

The Language Model: We consider initially *RTE_0*, a restricted subset of Real-Time Euclid (RTE) [8]. As in RTE, *RTE_0* programs consist of a static number of top-level processes and procedures, each of which is structured, allowing sequences of statements, conditionals, and loops. All loops are `for`-loops, with compile-time knowable loop bounds. There are no recursive calls nor dynamic data structures.

Unlike RTE, *RTE_0* has no exception handlers and no direct hardware operations. An *RTE_0* program contains at most one monitor shared by some or all processes. There are no conditional variables, waits, signals or broadcasts. The monitor may have multiple entries, however. Multiple monitor entries are modeled as a collection of critical sections of possibly different sizes, with the property that if a process is executing

*New Jersey Institute of Technology, Department of Computer and Information Science, Newark, New Jersey 07102
†Seton Hall University, Department of Mathematics and Computer Science, South Orange, New Jersey 07079

inside any one section, another process requesting entry to this or another section is blocked until the executing process exits the section. There is a separate processor for each process. All delays are either fixed, or associated with a wait for the critical section, and there is a mechanism for insertion of fixed delays. Procedures, other than top-level processes (and critical sections), are either called from a single process, or have no static variables, and do not access globals, nor use critical sections. Finally, all processes execute but once, all are available, along with their deadlines, at system startup time and all have no or fixed compile-time knowable start delays.

In addition, we make the following further assumptions for ease of presentation and without loss of generality: (1) all procedure calls have been inlined and (2) predicates of conditionals and loops do not use critical sections.

Symbolic Execution Assumptions: We also make assumptions on symbolic execution common to most static analyses. For schedulability analysis to give valid negative answers, and for our program transformations to apply, we assume: (1) every branch out of a conditional is either compile-time determinable as dead code or will be taken on some execution for appropriate input; (2) every combination of branch decisions determining a flow graph path through a process can be exercised by some appropriate input; (3) thus, except for the `for`-test, decisions taken in one iteration of a loop do not affect those taken in another (since we can view the sequence of decisions as determining a flow graph path in the unwound loop); and (4) decisions taken in one process do not affect the execution flow path through any other process. In future work, we will consider explicitly paired decisions, including loop-invariant conditionals, either in one process or across processes; implicit pairing of conditionals is much harder to detect and handle.

3 Static Analysis and Clustering

Compile-time analysis: There are two main techniques for compile-time analysis of program semantics: attribute grammars [1,3,7] and data flow analysis [2,3,4].

Data flow analysis techniques which may prove useful and will be considered in our integrated system, include dead-code elimination, exception elimination (including range-check elimination), code motion, constant propagation, and intelligent register allocation. Dead code elimination is deadline-preserving, while the others, if not implemented carefully, can be deadline-destroying (first, for example, because additional code that may have to added to implement the transformation may result in extra delays before a critical section is entered from a loop for the first time; second, because the critical section may be requested earlier on subsequent iterations, which, depending on the scheduler, may cause problems in other processes). In our system, we plan to study methods of implementing such "optimizations", and characterizing their applicability.

The principal transformations used in this paper are *clustering* transformations, easily implementable via attribute grammars. Intuitively, attribute grammars collect properties associable to language constructs by assigning values to fields of records (attributes) associated with the nodes of the parse tree. These values are computed and propagated through the tree one production at a time.

The Clustering Algorithm—An Intuition: Essentially, we can view a sequence of delay-free statements as a single statement taking as long as the sum of the lengths of the individual statements. Likewise, a conditional with two delay-free branches will certainly take no longer than the longer branch, and by above assumptions, could take that long. We can also condense a loop without delays into a node taking as long as the loop would have, that is, approximately, the size of the range times the sum of the length of the loop body and the time for incrementing the range variable. In a loop with delays, the delay-free tail of an iteration of the loop can be merged with the delay-free head of the next iteration.

We can, however, do more: we can insert fixed delays into flow branches to make accesses to a critical section happen at the same time as on other branches. The critical section will then start and finish at the same time on both branches. Inserted delays that are needed to balance a non-critical section node only block the process they are inserted in. On the other hand, inserted delays that are needed to balance a critical section node extend the shorter critical section, and thus not only block the process they are inserted

Figure 1. Part of An Abstract Grammar for *RTE_0*

Program	::=	Process Processes |
		Process
Process	::=	Stmt
Stmt	::=	ϵ |
		assign (var, Expr) |
		delay (symb_expr) |
		delay (symb_expr) Section |
		Stmt_1 Stmt_2 |
		if Test then Stmt_1 else Stmt_2 |
		for Range do Stmt_1

in but also other processes seeking entry to a critical section.

Moreover, we can insert varying delays into a shorter branch that does not access a critical section to mimic the critical section access done in a longer branch. The insertion of delays requires that during schedulability analysis as well as at runtime[1] when an inserted delay is reached the process acts and has the same effect on other processes as if it were trying to execute a critical section of the size of the delay. The insertion of appropriate delays is our principal deadline-preserving transformation.

Clustering does two things: first, it makes the resulting flow graph smaller and simpler (and thus faster to analyze); second, it transforms the program into one which is more nearly compile-time schedulable. The extension of power and expressivity over amalgamation in Real-Time Euclid comes primarily from the use of more sophisticated tests on conditionals.

A Grammar for *RTE_0*: The clustering algorithm uses an abstract grammar[2] for *RTE_0*, the interesting parts of which are given in Figure 1. ϵ is an empty statement. `Range` is a compile-time knowable range of integers, `Expr` and `Test` are expressions not involving critical sections. `Section` invokes a critical section, and may have parameters beyond the name of the calling process. *symb_expr* is (at compile time) a symbolic integer expression, whose origin and purpose will be explained later in this section.

Semantic Functions and Attributes: Figure 2 gives the attributes and semantic functions used by the clustering algorithm. Some of these attributes and functions, such as *sequence* and *new_delay*, are used only in more sophisticated rules for conditional.

Each *symb_expr* may involve (an arbitrary number of instances of) the *max* function, together with non-negative integer combinations, possibly including constant terms, of integer variables generated by *new_delay*. For instance, if d_1, d_2, d_3 are delays for calls to the critical section, one possible symbolic expression is given by

$$max\{d_1 + max\{d_3, 12\}, d_2 + d_3 + 2, 2d_3 + 7\}$$

Except for the conditional and loop statements, the attribute rules are mostly self-evident. For simple statements, *stmt.end* will be given by *stmt.start* plus its duration (given by *clock* plus delays); for other compound statements, such as sequence, by the *end* of its last component. For instance, for *assign*: *delay-free* and *combinable* are true, *sequence* is empty, *length* is 0, *start* is inherited, *end* is *clock(assign)+start*, *graph* is a single node labeled with *clock(assign)*, *clock* returns the amount of time it takes to compute the expression and do the assignment, and *new_delay* is not used.

Handling loop statements is conceptually easy, requiring one case when the loop body is delay-free, and one when it includes delays, but moderately tricky to specify precisely; details will appear in the full paper. There are a variety of tests on conditionals; these are described in the next section.

[1] That is, during both symbolic and real execution.

[2] A concrete grammar for *RTE* can be found in [15]. An *abstract grammar* ignores features irrelevant to the semantic aspects of interest, in this case, for example, declarations and keywords. The parse is given by the concrete grammar, but use of the abstract grammar makes the semantics easier to comprehend.

Figure 2. Attributes and Semantic Functions for Clustering

Attributes

name	type	description
delay-free	boolean	true iff there is no varying[3] delays in the argument
combinable	boolean	true iff can combine the argument with others, a tunable parameter
time	int	code execution time, excluding variable delays
sequence	int list	chunks of execution time between delays
length	int	the maximum number of delays on a path
start, end	symb_expr	schedule time
graph	flow graph	clustered flow graph for code

Semantic Functions

name	domain → range	description
clock	stmt → int	execution time for non-delay stmts
new_delay	→ symb_expr	generates a new integer variable

4 Clustering of Conditionals

There are at least three dimensions on tests for conditionals: (1) what structure (in particular, loops or conditionals) can the branches involve?, (2) what can the delay structure of the branches be?, and (3) how can we compare the times of the two branches?

We can easily (a) compare two delay-free branches, or (b) take any branch over an empty branch; a small extension of (b) lets us (b′) take a branch with arbitrary structure and time T over a delay-free branch with smaller time T'. In either case, we can insert delays; in the first case, the delays are constant, but in the second they can be symbolic, but still cannot result in a program which fails to satisfy a deadline if the original program (modulo symbolic execution assumptions) would.

Otherwise, our system will compare branches only if neither contains a conditional or loop statement which itself contains a delay, that is, if both branches are effectively linear.[4] In this case, we may be able to select one branch and insert delays in the other, to constrain calls to the critical section to occur exactly at times when calls would be made on the other branch, without violating a deadline.

We use two versions of such tests. The first (c) applies to branches of the same length, and compares the sequences of execution times. We say that one sequence *dominates* another if its elements are pairwise greater than or equal to the other's. If the sequence for the branch with greater total time dominates the other branch sequence, then delays can be inserted in a deadline-isomorphic manner. Our current algorithm assumes the sequences are sequences of known constants; a slightly more powerful version might allow comparison of sequences of *symb_exprs*.

The second (d) applies to branches of different lengths: if the longer also has the greater total time, and if the sequence of times for that branch can be partitioned (by adding consecutive elements together) so as to dominate the other sequence, then we can still insert delays in a deadline-isomorphic manner.

We illustrate these tests ((a), (b′), (c) and (d)) in Figure 3; the full paper will contain proofs of these claims. The i^{th} non-critical section straight-line node is represented as a circle labeled $s_i = x$ on the inside, where *clock(node)=x*. The i^{th} critical section straight-line node is represented as a square labeled $c_i = x$ on the inside, where *clock(node)=x*. The i^{th} delay prior to entering a critical section is represented as a black disk labeled d_i on the outside. All resulting delays are generated and inserted in the right branches. In (a), a fixed delay of 2 is inserted between s_3 and s_4. In (b′), a varying delay of $4 + d_1$ is inserted between s_3 and s_4. In (c), a fixed delay of 3 is inserted between s_3 and d_2, another fixed delay of 2 is prepended to c_2 and yet another fixed delay of 2 is inserted between s_5 and s_6. In (d), a varying delay of $6 + d_1$ is inserted between s_3 and d_3 and a fixed delay of 3 is inserted between s_6 and d_7.

5 Deadline-Extending Transformations

Although our tests can insert delays, and eliminate conditionals from schedulability analysis, some simple conditionals may remain *irreducible*, resistant to clustering. Consider, for instance the graph in Figure 4. The graph was obtained by taking the graph (c) of Figure 3 and setting c_1 and c_2 to 2 and 6 respectively. Now, the two branches remain different but neither dominates the other. Thus, clustering delay insertions are no longer guaranteed to be deadline-preserving. However, if we insert a fixed delay of 3 between s_1 and s_3 and prepend c_1 with a fixed delay of 4, then the new conditional is reducible (by inserting a fixed delay of 2 between s_5 and s_6). Unfortunately, the effect of the two insertions is not deadline-isomorphic or preserving, but is deadline-extending.

In general, deadline-extending transformations involve delay insertions on multiple branches. Another interesting class of transformations involves introducing only some steps of a complete deadline-extending transformation. In the same example, a simpler (than the original) graph with a partially-reduced conditional results if only the delay of 3 between s_1 and s_3 is inserted, but c_1 is not prepended with a delay of 4. This transformation, while potentially deadline-destroying, nevertheless preserves more of the original conditional

[4] We are assuming that only conditional branches which are linear are *combinable*. We will attempt to relax this assumption in our future work.

Figure 3. Four Conditional Clustering Tests

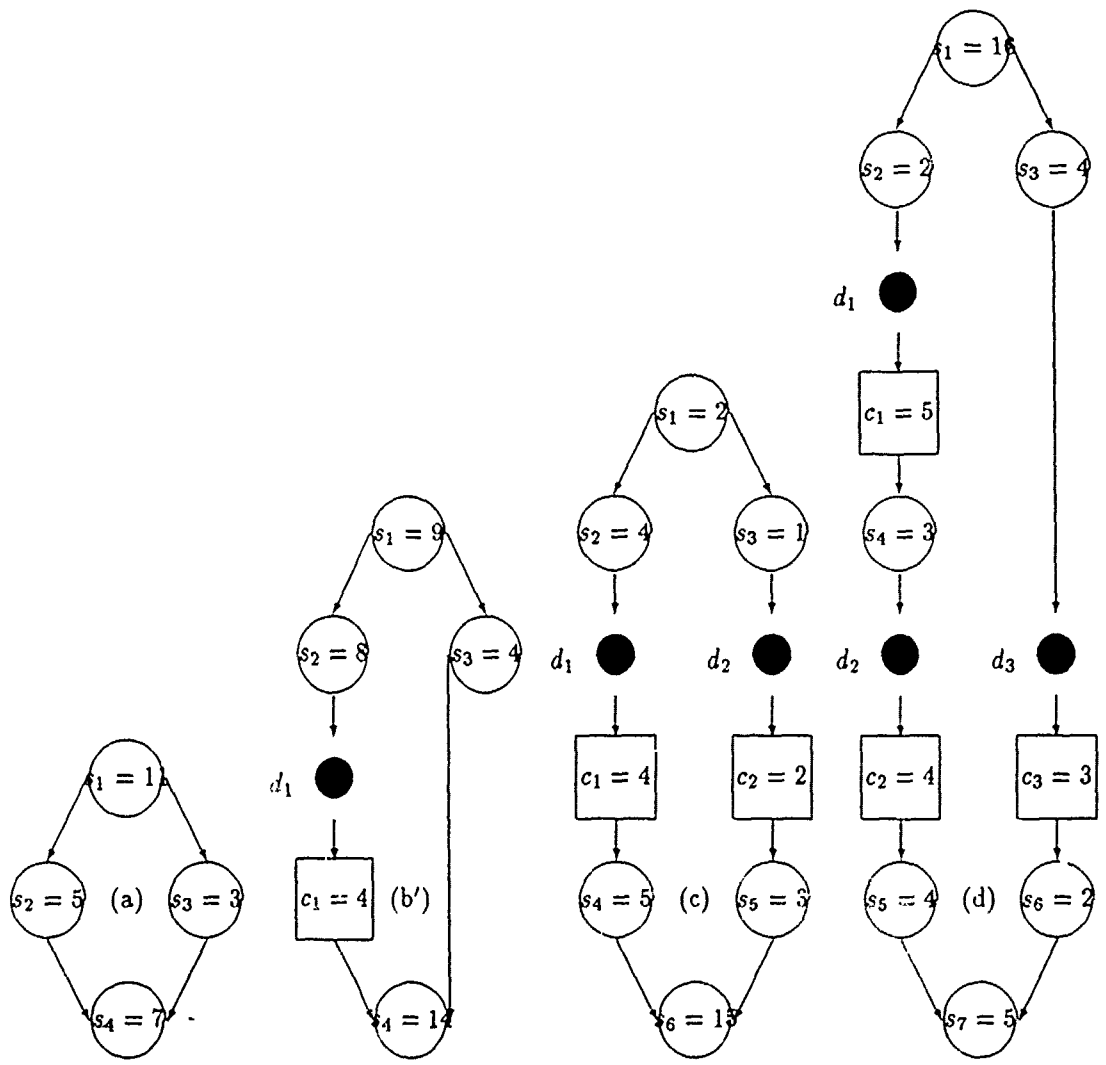
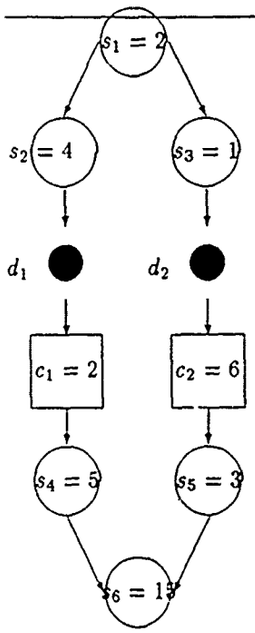


Figure 4. An Irreducible Conditional



structure. The transformation is of interest because we believe that even such partially-reduced graphs may already be analyzed for schedulability in polynomial time using incremental techniques.

While more work is needed to understand fully the effects of such deadline-extending and other related transformations, we have undertaken preliminary studies on random flow graphs modelling programs in *RTE_0* to determine how often these transformations will find a deadline-satisfying schedule for the original program. The results we have gathered so far are encouraging.

A Dual Scheduling Algorithm: The resulting, post-clustering program is, of course, much easier to analyze, than the original program. Furthermore, when a deadline is not satisfied, we can undo the deadline-extending transformations one at a time until the deadline is satisfied. In some sense, this is analogous to dual algorithms for linear programming: we begin with an optimal program, and iterate until we find a feasible one. The early steps of this algorithm will be quite inexpensive, since most processes will be represented by straight-line flow.

6 Future Work

Extend the Model: We intend to include more general language constructs. At the very least, we would like to re-introduce most if not all currently omitted RTE featues, including multiple monitors, different start times, periodic and aperiodic processes and so forth. We will also attempt restricting processors to a limited number.

Data Flow Analysis: We intend to categorize applicability of classical compiler optimizations to hard real-time programming, determine transformations and preconditions, and apply interprocedural analysis (in particular, aliasing).

Heuristics for Dual Scheduling: The choice of deadline-extending transformations to reverse will significantly affect both the number of such steps required and the cost of performing those expansions. We need to determine suitable heuristics, considering the amount and the difference of inserted delays on branches, their position in the flow graph, and the amount by which a given process fails to satisfy a deadline. In addition, we will consider the possibility of incremental analysis of scheduling as transformations are undone. The fact that schedules are monotonic under expansion (times can only decrease, since we can always keep the old schedule) may allow efficient incremental analysis.

References

[1] Deransart, P., Jourdan, M., Lorho, B., *Attribute Grammars: Definitions, Systems, Bibliography,* No. 323, Springer-Verlag, Lecture Notes in Computer Science, May 1988.

[2] Allen, F. E., Rosen, B., Zadeck, F. K., *Compilers, Optimization,* ACM Press/Addison-Wesley, (to appear), 1991.

[3] Aho, A. V., Sethi, R., Ullman, J. D., *Compilers: Principles, Techniques, Tools,* Addison-Wesley, Reading, MA, 1986.

[4] Hecht, M., *Flow Analysis of Computer Programs,* Elsevier Sequoia S.A., Lausanne, Amsterdam, the Netherlands, 1977.

[5] Halang, W. D., "A Proposal for Extensions of PEARL to Facilitate Formulation of Hard Real-Time Applications," *Informatic-Fachberichte 86,* pp. 573–582, Springer-Verlag, September 1984.

[6] Haase, V. H., "Real Time Behavior of Programs," *IEEE Transactions on Software Engineering,* pp. 494–501, SE-5, No. 7, September 1981.

[7] , Knuth, D., "Semantics of context-free languages," *Mathematical Systems Theory,* pp. 127–145, Vol. 2, No. 2, February 1968, Correction, 5 (1), 95–96, March 1971.

[8] Kligerman, E., Stoyenko, A. D., "Real-Time Euclid: A language for reliable real-time systems," *IEEE Transactions on Software Engineering,* pp. 941–949, SE-12, No. 9, September 1986.

[9] Puschner, P., Koza, Ch., "Calculating the Maximum Execution Time of Real-Time Programs," *International Journal of Time-Critical Computing Systems,* pp. 159–176, Vol. 1, No. 2, September 1989.

[10] Shaw, A. C., "Reasoning About Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering,* pp. 875–889, SE-15, No. 7, July 1989.

[11] Zedan, H., "On the Analysis of OCCAM Real-Time Distributed Computations," *Microprocessing and Microprogramming,* pp. 491–500, Vol. 24, 1988.

[12] Mok, A. K., Amerasinghe, P., Chen, M., Tantisirivat, K., "Evaluating Tight Execution Time Bounds of Programs by Annotaticns," *IEEE Workshop on Real-Time Operating Systems and Software,* Pittsburgh, PA, pp. 74–80, May 1989.

[13] Park, C., Shaw, A. C., "Experiments with a Program Timing Tool Based on a Source-Level Timing Schema," *IEEE Real-Time Systems Symposium,* Orlando, FL, December 1990.

[14] , Stoyenko, A. D., "A Schedulability Analyzer for Real-Time Euclid," *IEEE Real-Time Systems Symposium,* San Jose, CA, December 1987.

[15] , Stoyenko, A. D., *A real-time language with a schedulability analyzer,* University of Toronto, Department of Computer Science, Technical Report CSRI-206, December 1987.

[16] Shaw, A. C., *A Formal System for Specifying, Verifying Program Performance,* Carnegie-Mellon University, Computer Science Department, CMU-CS-79-129, June 1979.

[17] Shaw, A. C., *Determinstic Timing Schem : for Parallel Programs,* University of Washington, Department of Computer Science and Engineering, 89-05-06, May 1990.

PIPS: An Integrated Approach to the Design of Real-Time Systems [1]

Chien-Chung Shen and Rajive Bagrodia

Computer Science Department
University of California
Los Angeles, CA 90024

1 Introduction

In the paper, we present an *integrated approach* to the design of real-time systems[2]. The goal of this approach is to allow timing constraints of a proposed design to be evaluated in the early stages of system design and subsequently monitored through system implementation. The novel feature of this approach lies in its use of *Partially Implemented Performance Specifications* (PIPS) as the paradigm to both *model* and *design* real-time systems. A PIPS model is a partially implemented system where some system components exist as simulation models and others as operational modules. A PIPS model may be *iteratively* transformed into an operational system, while its timing properties are monitored during critical stages of model refinement. A language and its execution environment have been defined to support the use of PIPS models for system design. The language, called RTM, may be used to program a real-time system as well as its simulation model and is used to develop the initial model, its subsequent refinements and the operational system. RTM supports interrupts and task/message priorities, and may be used to program sporadic, periodic, and adaptive[MH89] tasks. An RTM program may be executed in an appropriate environment to evaluate its satisfaction of timing constraints under various workload.

Section 2 outlines the PIPS approach. The RTM language is briefly described in section 3. An example together with its experimental results is presented in section 4 to illustrate the applicability of the PIPS approach.

2 PIPS Approach

Traditional real-time system design methodologies suggest that a simulation model be constructed first in a special language. After evaluating the satisfaction of timing constraints, the model is then implemented entirely in a real-time programming language, like Ada. These approaches imply that system modeling and system design/implementation are carried out independently. Unlike other design methodologies, the PIPS approach uses the RTM language to write both a simulation model and its operational system, and suggests how an initial model may be transformed iteratively into an operational system where the design process is monitored from initial modeling to final implementation to ensure that it is consistent with the specification. The PIPS approach is based on the following concepts.

- **PIPS Model:** A PIPS model consists of both simulation steps and operational steps. For a simulation step, the notion of physical time is captured by the *simulation clock*, where the notion of *progress of physical time* is modeled by the increment of the simulation clock. When executing a simulation step, the simulation clock is incremented by the amount estimated to be the physical time required to actually execute the corresponding operational step. In contrast, an operational step is a sequence of actual program statements, where its physical time consumption is measured by a physical clock.

- **Logical Element:** The concept of *Logical Element* (LE) is introduced to facilitate the allocation of processes to processors. The simulation or operational steps of all processes mapped to a common LE are executed sequentially, and those mapped to different LEs are executed (logically) in parallel. Each LE is assigned its own *clock*, and will eventually be replaced by a physical target processor when the designed system is operational.

[1] Partially supported with a grant from Hughes Aircraft Co.-State of California MICRO Project and also by NSF (CCR 88 10376).

[2] We use the term real-time systems to include distributed real-time systems.

The key issue of the PIPS approach lies in its ability to integrate the execution of simulation steps together with operational steps, where both logical time and physical time are used to update the common notion of time to obtain overall timing characteristics. The PIPS design paradigm is summarized as follows. A complete description of the PIPS approach can be found in [BS90].

- Given the functional and timing specifications for a real-time system and an initial system design, an initial simulation model is developed. The satisfaction of timing constraints imposed on the system can be evaluated by executing the initial model under the proposed operating environment (workload).
- After being evaluated to satisfy timing specifications, the model is refined iteratively by elaborating a simulation step into an operational step, or by replacing a hardware model by the hardware itself. This suggests a PIPS model which consists of operational software and hardware components interspersed with logical models of other software and hardware subsystems. Note that in a PIPS model, the physical time consumed by an operational step will also be measured to update the corresponding clock. The refined model may again be executed to ensure that the specified timing constraints are satisfied under the proposed operating environment. This process is repeated until the model is transformed into an operational system.

3 RTM Language

The unique feature of the RTM language is its ability to program real-time systems as well as their simulation models. RTM is a process-based language derived from C. Simulation constructs are provided to support model development, while constructs to specify timing constraints, interrupts, process communication and synchronization are added to facilitate real-time programming. A complete description of the RTM language can be found in [BS]. The following are some of its language features.

- **Process Definition and Process-to-Processor Mapping:** The **entity** construct is used to represent processes. When created, an entity may be mapped to a specific LE and assigned a priority by using the **new** statement together with **on** and **priority** phrases.
- **Physical Time Modeling:** An entity executes a **hold** statement to model the progress of physical time, where the clock associated with the LE on which this entity resides is updated by the amount specified. A **hold** statement may subsequently be replaced by actual program statements in model refinements.
- **Process Communication:** Entities communicate with each other using buffered message-passing. The **invoke** statement deposits a message in the message buffer of the destination entity. Each message carries a timestamp and its scheduling information (described next). An entity receives messages from its message buffer by executing a **wait** statement, which specifies messages that are accepted by the entity and also defines their subsequent actions. Messages are received in the order of priority and timestamp.
- **Scheduling Information Specification:** RTM provides a new type as a general notation to describe scheduling information by which timing constraints for each individual activity initiated by a message may be specified. The information is then used by the scheduler to schedule the activities appropriately. This type, named **type_scheduler_info**, is implicitly defined by the language and consists of three fields, *priority*, *deadline* and *ptime*; *ptime* refers to the processing time needed for the activity, *deadline* refers to the deadline for completion of the activity and *priority* refers to the priority of the message.
- **Interrupt Specification:** The **interrupt** phrase is used to denote an interruptible computation step (either simulation or operational), where interrupting messages are specified together with their interrupt service actions.

4 An Example: Modified Landing System

A centralized prototype PIPS environment has been implemented at UCLA on a Sun 3/60 workstation. The environment consists of the RTM language compiler and the PIPS runtime system.

We use a modified version of the Martian Lander[JM86] to illustrate the design of a real-time system by the PIPS approach, where the satisfaction of a timing constraint is evaluated at different stages of its development. The lander entity repetitively invokes an I/O entity (`i_o`) to read the acceleration which takes a constant time (80 time units) to complete. In addition, we assume that the I/O entity shares a non-preemptible processor with a navigation entity (**nvg**) which is repetitively requested by a pilot entity to perform some navigation function. As a result, the start of an I/O operation may be delayed and thus even though it takes less than 100 time units, it may not be able to complete within 100 time units. Figure 1 shows the simulation model written in RTM. Note that both I/O and navigation entities are allocated to a common I.E, `le7`, and therefore are executed sequentially. In addition, they are initiated by lander and pilot entities respectively using **invoke** statements, and **hold** statements are used to model the I/O and navigation operations.

We then refine the preceding simulation model into a PIPS model by elaborating the **hold** statement of the I/O entity into operational code. Note that the code for the I/O entity changes only to the extend that actual statements, represented by function `io_action`, is included in the entity, as shown in figure 2. The PIPS runtime system measures the physical time consumed by the I/O operation to support overall timing measurements. The refinement process can proceed further until the entire system is operational, where the timing characteristics is evaluated at every stage of model refinement.

We now present the performance measurements for the modified landing system, where both the percentage of missed I/O deadlines and the average I/O response time are measured at the following three stages of model refinement. In stage one, the system exists as a simulation model. For the first experiment (figure 3(a)), I/O and navigation operations are performed alternatively. The time taken by a navigation operation is sampled from an exponential distribution. The metrics are plotted as functions of the mean navigation time. As seen from figure 3(a), the percentage of I/O misses increases monotonically as the mean navigation time increases. Also, the average I/O response time is prolonged by the amount equal to the mean navigation time. For the second experiment, the mean navigation time is fixed at 20, and the navigation activity is initiated for a percentage of the I/O activities. Let p be the probability that a navigation activity will be initiated together with an I/O activity. Figure 3(b) plots the metrics as functions of p. As seen from the figure, both the percentage of I/O misses and the average I/O response time increase as the probability p increases. In the second stage, the I/O entity executes an operational step to perform an I/O action. The physical time required to perform an I/O operation is generated by using a **for** loop. To validate the PIPS approach, the physical time consumed by the **for** loop on a Sun 3/60 workstation is kept equal to 80 time units. In the last stage, the simulation step for the navigation entity is also elaborated into an operational step. The above two experiments are repeated for these two stages, and their results are plotted on the same figure. As seen from the figure, the measurements are in reasonable agreement for each of the three stages.

References

[BS] R. Bagrodia and C.-C. Shen. Integrated design, simulation and verification of real-time systems. *to appear in Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.

[BS90] R. Bagrodia and C.-C. Shen. MIDAS: Integrated design and simulation of distributed systems. Technical Report, CSD-900027, Computer Science Dept., UCLA, September 1990.

[JM86] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

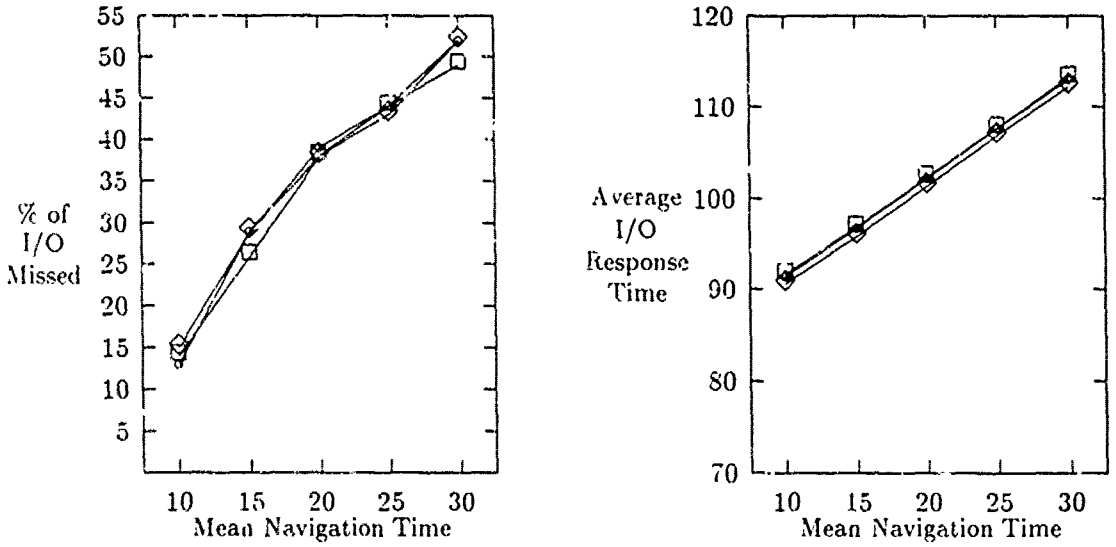
[MH89] A.H. Muntz and E. Horowitz. A framework for specification and design of software for advanced sensor systems. In *Proceedings of 10th Real-Time Systems Symposium*, pages 204–213, Santa Monica, California, December 1989.

```
#define NO_OF_IO   1000                    /* # of I/O requests */
#define NO_OF_NVG  1000                    /* # of navigation requests */

entity driver{ }
{  e_name io, ld, nv, pl;  le_name le7, le3, le4;
   io = new i_o{ }       on le7;           /* I/O entity */
   nv = new nvg{ }       on le7;           /* navigation entity */
   ld = new lander{ io } on le3;           /* lander entity */
   pl = new pilot{ nv }  on le4;           /* pilot entity */
}

entity lander{ io }
e_name io;
{ clock_type ss, tt;  message ioint;  int i;
  for (i = 0; i < NO_OF_IO; i++) {
     ss = current_clock();
     invoke io with begin { self };        /* start I/O request */
     wait until mtype(ioint) {             /* I/O complete */
       tt = current_clock();
       printf("I/O takes = %d seconds.\n", (tt - ss)); } /* I/O response time */
  }
}

entity pilot{ nv }
e_name nv;
{ message complete;  int i;
  for (i = 0; i < NO_OF_NVG; i++) {
     invoke nv with start{ self };         /* start navigation request */
     wait until mtype(complete);           /* wait for its completion */
  }
}

entity i_o{ }
{  message begin { e_name lander; };
   for (;;)
     wait until mtype(begin) {
        hold(IO_TIME);                     /* a simulation step for I/O action */
        invoke msg.begin.lander with ioint; /* I/O complete */
     }
}

entity nvg{ }
{  message start{ e_name pilot; };
   for (;;)
     wait until mtype(start) {
          hold(exp(MEAN_NVG_TIME));        /* a simulation step for navigation */
          invoke msg.start.pilot with complete; /* navigation complete */
     }
}
```

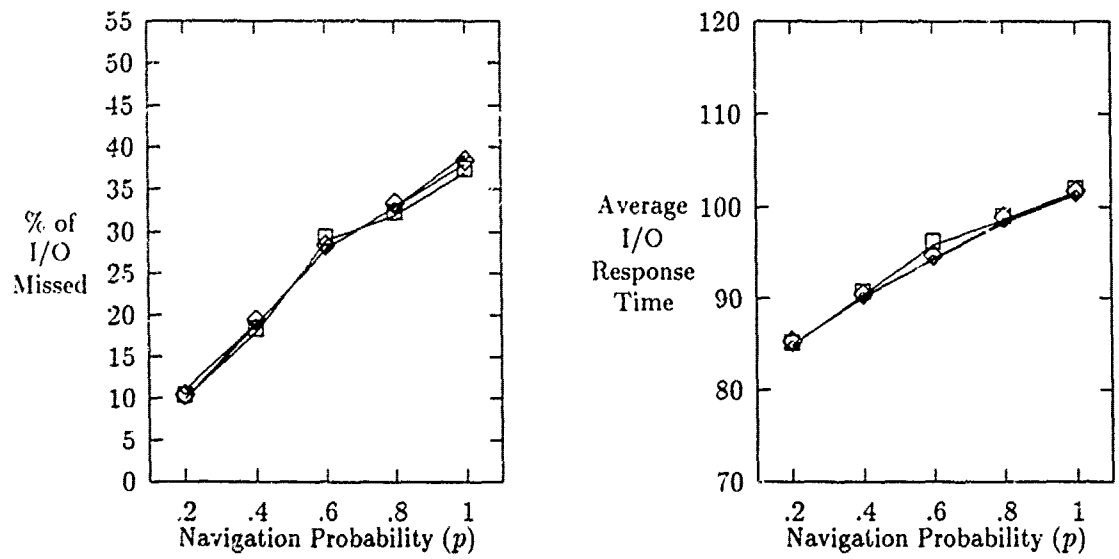
Figure 1: Modified landing system simulation model.

```
entity i_o{ }
{  message begin { e_name lander; };
   for (;;)
     wait until mtype(begin) {
        io_action();                          /* an operational step for I/O action */
        invoke msg.begin.lander with ioint; /* I/O complete */
     }
}
```

Figure 2: Modified landing system PIPS model: operational I/O entity.



(a) Alternative case.



(b) Probabilistic case (mean navigation time = 20).

Figure 3: Performance metrics for the modified landing system.

Graphical Prototyping of Tasking Behaviour

R.Lintulampi, P.Pulli
Technical Research Centre of Finland (VTT)
Computer Technology Laboratory
P.O. Box 201 SF-90571 Oulu Finland
E-mail: tkorli@finvtt

Abstract

The Espex tool is a graphical simulation and prototyping tool for Structured Analysis for Real-Time Systems (SA/RT). Petri net like Espex graphs are used for modeling a system. The execution principle for a logical model and a task model is described. Pre-emptive priority based scheduling is simulated in the execution of a task model. Experience in using the Espex tool on real examples are also reported. The Espex tool has been implemented in the Smalltalk-80 programming environment and it runs on Sun-3 and MacIntoshII workstations.

1 Introduction

Prototyping of software systems has been suggested as an alternative approach for software development [Vonk90, Hekmatpour&88, Connell&89]. The prototyping approach can be used with the conventional waterfall model of software development. In recent years a new software development concept, the spiral model, has been presented where prototyping is a part of the development process [Boehm88, TRW89]. Recently, interest in using prototyping in the development of real-time system has increased [Gabriel&89].

A prototype is an executable model of a system. In the early phases of software development prototypes can be built directly by using an executable graphical specification language. Prototypes can be divided into behavioural prototypes and structural prototypes [Gabriel&89]. Behavioural prototypes define what the system is supposed to do. Structural prototypes define how the system will accomplish its behaviour. An ideal prototyping environment supports both behavioural and structural prototypes. Prototypes must be quick to be built and modified and easy to gather data during about their use.

In the field of real-time embedded systems a group of modeling languages, known as embedded behaviour languages has proven very useful. Well known examples of these languages are Ward&Mellor [Ward&85], Hatley&Pirbhai's Structured Analysis extension [Hatley&87] and Harel's Statecharts [Harel87]. Recently tools which also support the execution of these languages have been reported [Harel&90, Athena89]. These tools support only the requirements phase of software development. In this paper we present the Espex tool which supports both behavioural and structural prototyping of software specifications and designs.

2 The Espex Tool

2.1 Espex Graphs

The ESPEX tool is a graphical simulation and prototyping tool for Ward&Mellor's SA/RT. It has been developed from the SPECS tool originally designed at ETH in Zürich, [Dähler&87, Pulli88] and it has been implemented in the Smalltalk-80 programming environment on Sun-3 and MacIntoshII workstations. A logical model, a processor model, a task model, and combinations of these models can be simulated in the Espex tool.

The Espex tool uses Espex graphs which are based on combining channel/instance Petri nets, predicate/transition Petri nets, and the object-oriented paradigm. SA/RT and Espex graphs make up a hybrid method which is at the same time communicating, flexible, formal, and executable.

The modeling elements have several attributes which differentiate the behaviour of the elements in the execution of different models. The continuous attribute specifies if an S-element represents a propagating or non-propagating element. The arrival of a data token at a propagating element activates the transformation connected to it. Signals and discrete data flows are propagating elements but stores and continuous data flows are not. A delay defines how long a data token has to stay in an S-element before it is available for its next transformation. A delay simulates the communication time that is independent from the simulated execution resource. A delay inside the simulated execution resorce is simulated by giving extra processing time to transformations. Capacity defines how many data tokens an S-element can carry at the same time. The processing time of a transformation is an estimate of how long the execution of the implementation takes in the target environment.

Each data token and stimulus consists of some data and a unique time stamp. Data can be any Smalltalk object. A special case of data tokens is a future token. The time stamp of a future token is not constant as a scheduler can modify it. For example, all tokens produced by simulated software are future tokens.

2.2 User Interface

The Espex tool consists of a model editor and a model simulator. It has been implemented on the basis of windows, pop-up menus, and a mouse pointing device. The simulator has been integrated into the model editor. At any time during the modeling process, the model can be simulated.

2.3 Simulation Time

The present Espex tool supports only the simulation of central clock systems, by using global simulation time. The simulation time has not been bound to any real time clock, because we did not find a convenient way to scale the real-time clock in different simulation situations.

3 Execution of Logical Model

The implementation free user-observable requirements for the system are the basis for a logical model. It describes the behaviour to be implemented by the specified system. The portions outside the specified system are included in terminators.

The main objective of the simulation of a logical model is to find out how a system to be modelled should work. The executable logical model constitutes a behavioural prototype of a system.

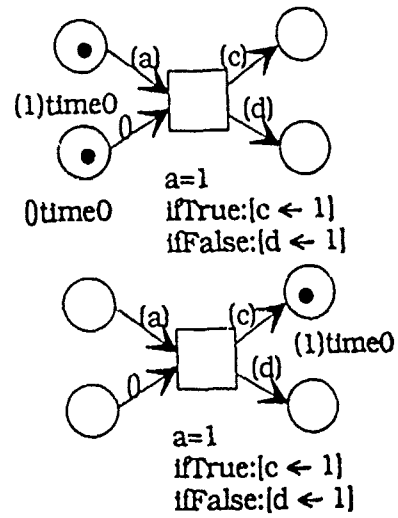


Figure 1. The minispecification is executed when the transformation is fired. Depending on the minispecification, a data token may be produced for a single S-element.

Execution rules for a logical model are based on the combination of the transformation schema presented in [Ward86] and Petri net behaviour. Basic rules are:

- All input flows must have a data token.
- Transformation will be fired when a data token arrives at a discrete data flow or a signal flow.
- After the firing, a data token is produced into flows specified by a minispecification.
Figure 1 illustrates the firing principle.

4 Execution of Task Model

A task model simulated in the Espex tool is based on a redistributed logical model to which physical characteristics have been added. A task is an independently schedulable piece of software which implements transformations assigned to it. Tasks are executed concurrently by an operating system in a target environment. Transformations allocated to a task are executed sequentially. The graphical presentation of a task model resembles that of a logical model. An executable task model constitutes a structural prototype of a system.

Simulation allows the testing of several diferent task allocations to find out the effects they have on a system's behaviour. Simulation can be divided into three phases: (1) validate priorities, synchronization and logical correctness of task interfaces, (2) find requirements for the implementation and (3) estimate the timing and complexity of a task model.

The following functions of a real-time operating system are simulated in the Espex tool:
- Priority based preemptive scheduling.
- Signals(semaphores).
- Mailboxes.
- Interrupts.
- Timers.

5 From Logical Model to Task Model

5.1 Allocation of the Logical Model to Tasks

The user can freely choose how to allocate transformations to different tasks and interrupt handlers. Interrupts can take place at any time and they are served immediately. Espex does not offer a methodology for the allocation but a designer can do the allocation as he wants. After allocation, the model is reorganized to make it more readable and priorities are set to tasks and interrupt handlers. The allocation and reorganization mechanism has been described earlier in [Pulli89].

5.2 Execution Order Numbers

The execution of a task model is deterministic. The order of the execution of the transformations within each task must be specified explicitly in the allocation by giving execution order numbers to the lowest-level transformations.

A task is an infinite loop in the Espex tool. The task's entry point is the point where the execution starts after activation and where the execution is returned after all possible paths are executed within the task. The task can have only one entry point.

5.3 Processing Times

Processing time has to be given to each lowest-level transformation in a model before the simulation. It is an estimation of how long the execution of the implementation of the transformation takes in a target environment. Processing time is relative to the simulation time.

5.4 Timers

Timers can be simulated in two ways: (1) using delay operations with flows and (2) using simulated hardware timers. A delay operation is useful in situations where there is no need to reset the timer. Otherwise a simulated hardware timer is needed.

6 Modeling Terminators

Terminators are presented as black boxes in SA/RT. Event lists describe the interaction between a model and terminators. The description of the interaction has to be precise and executable in the simulation of a model. This can be done in three ways in the Espex tool:

- Terminators can be modelled in the same way as the system is modelled. This is a useful way when terminators are simple or when they communicate interactively with the model.
- The user can give all input data before a simulation session or the input data can be read from a file.
- The user can give data interactively during a simulation session.

7 Scheduler

There is a two-level scheduling mechanism in the Espex tool. The system scheduler maintains a set of allocated processors and logical units (Figure 2). It randomly chooses an unit from the set and activates the unit scheduler. The unit scheduler of a simulated processor has a task list which has been sorted in the order of priority. The unit scheduler goes through the task list until a firable transformation is found or it notices that the execution of an earlier activated task is not yet finished. After that the control is returned to the system scheduler. All allocated units are scheduled after each increment of the simulation time. The new simulation time is the lowest value of time stamps of all data and future tokens in the system to be simulated.

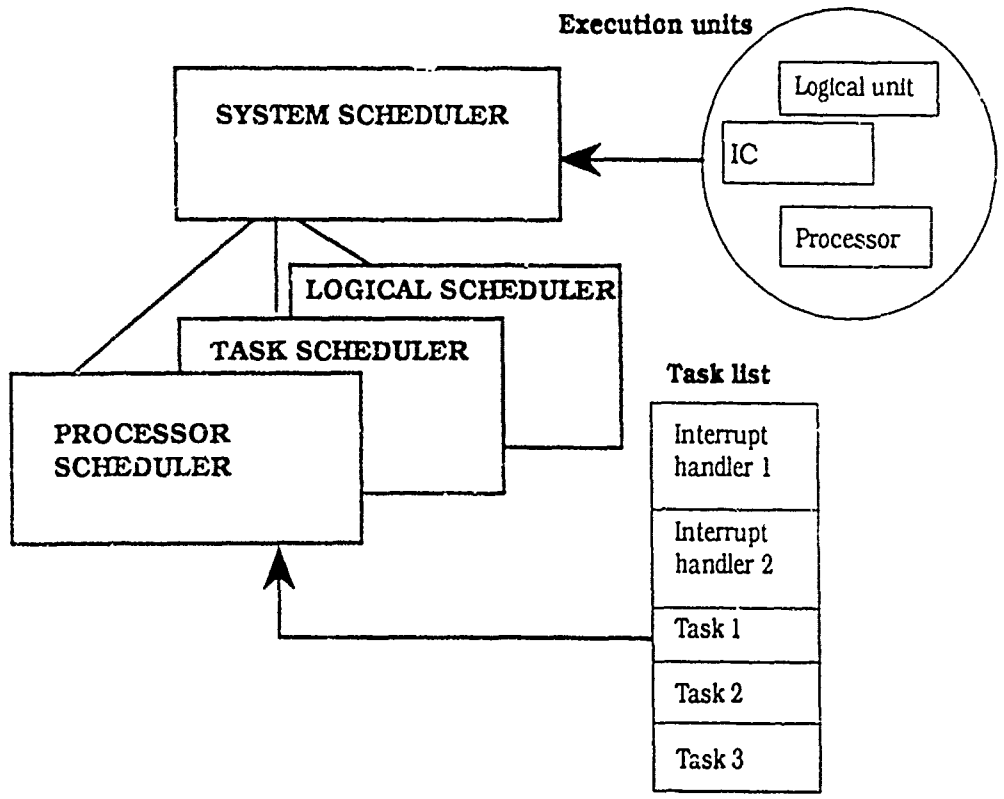


Figure 2. Two-level scheduling mechanism is used by the Espex tool

8 Experiences

The Espex tool has been used to validate the specification of a complex robot system. Only the control part of the specification was

modelled in the Espex environment. The control system consisted of twelve data flow and state transition diagrams. The Espex tool was found to be very suitable for validating complex control sequences. It will also be used in the validation of a processor model and a task model later in the project.

The Espex tool has also been used in modeling a specification of one OSI protocol layer and a task model of that model [Heikkinen90]. The original specification had been made using a commercial SA/RT-tool in a real industrial project and it was not executable. It consisted of 14 data flow and state transition diagrams and 30 minispecifications. The executable Espex specification consisted of 80 data flow and state transition diagrams (176 lowest level transformations), from which 28 diagrams described terminators not modelled in the original specification. The effort of converting the non-executable specification to an executable one was one man-month.

The following conclusions can be derived from the experiences:
- Executable models are very well suited for a validation of complex systems.
- The graphical simulation of different models indicated the correct behaviour of the models.
- The effort to convert a draft logical model to a strict executable model may be quite large.
- The Espex tool supports the work to be carried out in the analysis and design phases very well.
- When time is included in modeling elements, and well defined semantics are used in task interfaces and inside tasks, we found it straightforward to derive different executable task models from an executable logical model.

References

[Athena89] Athena Systems Inc., "Foresight: Modeling and Simulation Toolset for Real-Time System Development," User's Manual, March 1989.

[Boehm88] B.W.Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer Surveys, May 1988, pp.61-72.

[Connell&89] J.L.Connell, L.Shafer, "Structured Rapid Prototyping - An Evolutionary Approach to Software Development," Prentice-Hall Inc., New Jersey, 1989.

[Dähler&87] J.Dähler, B.Gerber, H.-P.Gisiger, A.Kündig, "A Graphical Tool for The Design and Prototyping of Distributed Systems," ACM Software Engineering Notes, vol. 12, no 7, 1987, pp. 25-36.

[Gabriel&89] R.P.Gabriel(editor), "Draft Report on Requirements for a Common Prototyping System," SigPlan Notices, vol. 24, no 3, March 1989.

[Harel87] D.Harel, "StateCharts: A Visual Formalism for Complex Systems," Science of Computer Programming 8(1987), pp.25-36.

[Harel&90] D.Harel, H.Lachover, A.Naamad, A.Pnueli, M.Politi, R.Sherman, A.Shtull-Trauring, M.Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," IEEE Transactions on Software Engineering, vol. 16, no 4, 1990, pp.403-413.

[Hatley&87] D.Hatley, I.Pirbhai, "Strategies for Real-Time System Specification," Dorset House 1987.

[Hekmatpour&88] S.Hekmatpour, D.Ince, "Software Prototyping, Formal Methods and VDM," Addison-Wesley Publishers Ltd, Wokingham GB, 1988.

[Heikkinen90] M.Heikkinen, "The Prototyping of Executable RTSA-Models," diploma thesis, University of Oulu, 1990, (in Finnish).

[Pulli88] P.Pulli, "Execution of Structured Analysis Specifications with an Object Oriented Petri Net Approach," Proc. ICCL'88 Conference, 1988, Miami Beach, Florida, pp.286-293.

[Pulli89] P.Pulli, "An Object Oriented Approach to Distributed Prototype Execution of SA/RT Specifications," Proc. STA5 Conference, Chicago, Illinois, 1989, pp.80-91.

[TRW89] "Process Model for High Performance Trusted Systems in Ada," TRW Systems Division, Fairfax, USA, 1989.

[Ward&85] P.Ward, S.Mellor, "Structured Development for Real-Time Systems," Volume II, Yourdon Press, New York, 1985.

[Ward86] P.Ward, "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," IEEE Transactions on Software Engineering 12(1986)2, pp.198-210.

[Vonk90] R.Vonk, "Prototyping - The Effective Use of CASE Technology," Prentice-Hall Ltd, London, 1990.

APPLICATION OF REAL-TIME SCHEDULING THEORY TO MULTIPROCESSOR PIPELINES

Robert J. Fornaro & William D. Allen
Precision Engineering Center
North Carolina State University

ABSTRACT

Cyclic serial computational applications can be decomposed into multiple tasks for implementation on a multiprocessor system. In a hard real-time system this is usually for the purpose of meeting computational deadlines. Such a decomposition results in a pipelining of the computation with each processor being a stage of the pipeline. A shared resource (memory, bus) is utilized for inter-task communication. This study reviews the applicability of well-known real-time scheduling protocols for deadline analysis in such a system. Additionally, the impact of tasks whose execution time is data dependent is addressed. Given the timing characteristics of the task set a priority assignment for resolving conflicting requests for the shared resource can be determined. Further, this analysis gives a method for establishing or evaluating the validity of the application's cyclic deadline.

THE APPLICATION

A real time machine control application can be abstractly described as: read data from a sensor, perform a series of calculations on that data, and output results. If this is a periodic process operating at a high frequency and the combined input/output and computation time is greater than that available for a given periodic requirement this computation will not meet its deadlines. Two solutions are possible: 1: use a processor fast enough to accomplish the application within its time constraints, or 2: divide the application across multiple processors achieving the needed speedup. The first solution is obvious and in principle trivial. The second must be carefully considered. If an initial delay can be tolerated, the application can be divided into several steps and the process pipelined thus overlapping portions of the calculation. Thus the periodicity of the application will be determined by the step having the longest duration. Dividing the application into tasks (steps) has its own set of ramifications. In particular, the issue of the exchange of data from one step to the next in the pipeline must be considered. In this study a blocking data passing protocol was utilized.

In the above scenario, assume a one task (step) per processor configuration. Then there is no intra-processor task scheduling required for this system. An implicit scheduling of task requests is created by the periodic scheduling of the data acquisition cycle and by the blocking nature of the inter-task communications. Since the assumed configuration uses a common resource (bus) for interprocessor communications the scheduling requirement is for this resource. Without some scheduling regime in place, it is possible for one task to unduly delay another by seizing the common resource thus causing a deadline failure.

SCHEDULING

Scheduling of a common resource implies the existence of either a scheduling/allocation entity separate from the processing nodes of the system or some form of inter-processor negotiation [1]. It is assumed that the latter is too costly in overhead and thus to be avoided in this case. If an a priori scheduling regime can be established, resource scheduling could be accomplished by an intelligent hardware arbiter or a runtime environment. Since a resource, not a task, is being

scheduled the hard real-time scheduling regimes normally used for task scheduling do not directly apply. They may, however, provide guidance in the development of a resource scheduling protocol for these applications. The most common scheduling scheme for hard real-time systems is the rate monotonic approach where priority is based on the periodicity of each task (with the shortest period receiving the highest priority). In the application considered here, the pipelining of tasks creates a situation where all tasks have the same periodicity. In this situation, rate monotonic scheduling reduces to a FCFS protocol since there is no basis for differentiating between tasks. Thus alternatives must be considered. Two dynamic scheduling protocols often considered are earliest deadline [2] and least slack [3].

With an earliest deadline scheduling protocol, what constitutes a deadline must be determined. Since all tasks have the same period, the only deadline criteria is that the task must be ready to exchange data at the next scheduling point. While earliest deadline is a dynamic scheduling algorithm, the fact that all tasks have the same period means that all task requests occur at the same relative point in time within each cycle resulting in deadlines which are fixed within the execution cycle. Since there is no dynamic behavior of the deadlines, this scheduling problem may be solved statically.

Least slack, while also normally a dynamic algorithm, can likewise be addressed as a static problem due to the common periodicity of the set of tasks. Since each task has a fixed period and all periods are identical, the slack times of each task are fixed. Thus given the execution time of each task and the required system periodicity, the slack times can be established and priorities established. Here of course the highest priority will be given to the task with the least slack. Since slack is readily determined and is directly related to establishing the magnitude of delay which can be accommodated by each task, it becomes the logical criteria for setting priorities for resource allocation.

Even with a resource scheduling regime in place, the problem of priority inversion caused by blocking delays can still occur just as in uniprocessor systems [5], [4]. As an example of how priority inversion can affect the applications being studied, consider the following example. A uniprocessor implementation of a certain application performs the following sequence of operations:

1. Read an input value x from a sensor.
2. Perform computations requiring 25 units of time.
3. Write adjusted values x' and y to controllers.

Assuming that input and output operations take one time unit each, this task requires 28 time units for each cycle.

If the application requires a sampling rate greater than one sample every 28 time units, it is possible to divide the application across multiple processors with each processor performing part of the computation. Assume the following partitioning of the application across four processors (P1 - P4):

P1: Input of x and first 6 time units of computation.
P2: 12 time units of computation.
P3: 4 time units of computation and output of x'.
P4: Remaining 3 time units of computation and output of y.

This decomposition provides a pipelining of the computations to achieve the needed speedup. (It should be noted that this arrangement assumes that there is no requirement for the results to be output before the next sample is taken.) Since data must be passed from one processor to the next, interprocessor data exchange points must be inserted in the tasks. If interprocessor communication is blocking, then there is an implicit synchronization between processors which occurs on each data exchange. Assuming that each data exchange takes 2 time units, then the maximum processing rate of this system becomes one sample every 16 time units (by P2). Figure 1 depicts this multiprocessor computation scenario.

P1	P2	P3	P4
read x	exch a	exch b	exch c
6	6	4	3
exch a	exch b	write x'	write y
	6	exch c	

(9 units) (16 units) (9 units) (6 units)
[Data exchange = 2 units]

FIGURE 1 - Multiprocessor Computation

With this implementation, after a startup delay the system would be expected to operate with a cycle of 16 time units. However, analysis of the timing of this application shows that after the startup delay the system produces results every 17 time units instead of the 16 units expected. Figure 2 shows this analysis.

```
P1: r======aar======aaaaaaaaaar======aaaaaaaaaar======aaaaaaaaa
P2: aaaaaaaaa======bb======aaa======bb======aaa======bb======aa
P3: bbbbbbbbbbbbbbbbb====wccbbbbbbbbbb====wccbbbbbbbbbb====wccb
P4: ccccccccccccccccccccccc===wccccccccccccc===wccccccccccccc=
```

[xxx: waiting for synchronization, xxx: data exchange,]
[r: data input, w: data output, ===: computation]

FIGURE 2 - Multiprocessor Timing Analysis

Inspection of the timing lines for P2 and P3 show what has happened. The request for data exchange between P3 and P4 (exch C) occurs 1 time unit before the request for a P1->P2 (exch A) exchange. Since no other request is pending the exchange begins and lasts for 2 time units. The data exchange process is indivisible so the P1->P2 exchange must wait until the P3->P4 exchange completes. Since there is no slack time in P2's cycle, this delay causes a missed deadline. Assuming that timings are continuous rather than discrete as in the analysis, it can be seen that P2 will be delayed if P3's computation time lies between 3 and 5 time units. If P3's time is 3 time units or less, the P3->P4 data exchange will be complete before the P1->P2 request. Likewise if P3's time is 5 units or more the P1->P2 exchange will get priority. Thus the delay can be as much as the length of time for the data exchange, in this example 2 time units. Therefore for the system to be intrinsically deadline safe either the deadline cycle must be set to 18 time units (giving P2 a slack time equal to the worst case delay) or P3 must be designed such that it can never execute in the 3 - 5 time unit range.

As can be seen from the example above, allowance must be made for the potential blocking of a task causing a deadline failure. Since execution of these tasks is deterministic, one solution is to adjust (lengthen) the execution time of the task creating the blocking delay so that the 'critical' task is not delayed. In this example, simply lengthening the task in P3 by one time unit would cause the requests for the P1->P2 exchange and the P3->P4 exchange to occur simultaneously. The priority scheme will then give resource access to the P1->P2 exchange which will thus allow P2 to continue without delay.

In performing an analysis of the blocking patterns of a task set, the possibility of the occurance of 'chains of blocking' must be considered. That is to say that the blocking delay of one task could cause that task to create a blocking delay on another task. Likewise, the adjustment of the

execution time on one task to eliminate a critical blocking could cause a blocking delay to appear elsewhere. Thus the blocking analysis/execution time adjustment must be an iterative process.

While there is a relatively simple solution to the blocking delay problem when task execution times are fixed, consider what happens when task execution times are data dependant. Consider the scenario presented above with the execution time of the task in P3 varying between 2 and 7 time units. As note before, delay of P2's task occurs only when the computation time of P3's task falls between 3 and 5 time units. As noted before, one solution to this problem would be to make the execution time of the task fixed at a value outside the timing region which will cause delays. This solution implies the necessity of having the capability of analyzing and establishing the timing of all possible paths through the task code.

Solution of the delay problem when tasks have variable execution times is simplified by the constrained nature of this particular application, primarily by the fact that all tasks have the same periodicity. As noted before this allows a static solution to the resource scheduling problem to be formulated. In the example above, there are several potential solutions. One is to insert delaying code into the task so that it will never execute in the 3 to 5 time unit range. This implies that a complete path analysis can be accomplished and that the code allows such an insertion. Another solution would be to have tasks schedule data exchanges according to a clock time relative to the initiation of the task's cycle. In the example above, if P3's task is not allowed to initiate the P3->P4 exchange until 9 time units after initiation it will not cause a delay to occur in P2's task. A third approach would be to have an intelligent arbiter allocating the resource. Since there is a fixed pattern of synchronizations which allows the critical task to meet its deadline, this intelligent arbiter would allow resource allocations only in the order specified by the desired execution pattern.

While each of the approaches shown above provide a solution to the delay problem, they are awkward and restrictive. Now consider a different approach. The previous solutions are predicated on preventing a delay of the critical task in the pipeline. Instead, suppose delays are allowed to occur and a way is found to bound the worst case delay of the critical task. Then the pipeline cycle will be established by the execution time of the critical task plus the worst case blocking delay caused by other tasks. Now it becomes necessary to determine and control the worst case blocking time. The example above shows that a delay equivalent to the time required for a single data exchange is possible. Further delays created by preemptions or chains of delay are also possible. The priority ceiling protocol [4],[5] limits the blocking of a task to the duration of one (the longest if they are not equal length) critical section and asssures that deadlock will not occur. (In relating to these works, note that the shared resource considered here is directly equivalent to a critical section. i.e. use of the resource is not preemptable.) Since the application is implemented as one task per processor and there is only a single resource being shared, deadlocks are not an issue and chains of blocking are not possible. Thus the maximum delay which can be imposed on any task in the system is that which can be caused by a single usage of the resource by a task of lower priority. Additionally, application of the concept of a priority ceiling establishes that the priority of a data exchange should be the maximum of the priorities of the sending and receiving tasks.

Due to the constraints created by this application decomposition, many of the commonly used scheduling and synchronization techniques either do not apply or have only limited applicability. In particular, the following results have evolved:

1. Due to the fixed cycle of the pipeline structure of the multiprocessor implementation of the application, dynamic scheduling algorithms reduce to a static scheduling problem.
2. Since all tasks have the same period, rate monotonic scheduling reduces to a FCFS protocol.
3. Since only a single resource is involved, priority inheritance protocols provide no improvement in avoiding delays.

4. Establishment of priorities for data exchanges should be based on the least slack principle.

What has thus been shown is that an application which can be decomposed into a pipelined implementation can be structured such that hard deadlines can be assured. For the task set of such a decomposition, a resource allocation priority assignment can be established which will assure that the critical (pacing) task of the pipeline will meet a deadline equal to its maximum execution time plus the maximum resource lock time by any other task. This scheme accommodates the problems of tasks with data dependent execution times without necessitating special treatment of those tasks. The resource allocation priorities of tasks are set using least slack as the criteria. Physical implementation of such a system does not require a separate synchronization processor or use of an intelligent arbiter for the resource. Simple fixed priorities will be sufficient.

CONCLUSIONS AND FUTURE WORK

A cyclic serial computational application can be decomposed into multiple small tasks for implementation on a multiprocessor system with the goal of reducing the cycle of computation. In a hard real-time system this is usually for the purpose of meeting computational deadlines. Such a decomposition results in a pipelining of the computation with each processor being a stage of the pipeline. Communication of results from stage to stage of the computation is accomplished through a shared resource. This study has reviewed the applicability of well-known real-time scheduling protocols to the problem of guaranteeing deadlines in such a system. The pipelined nature of this implementation produces a constraint which significantly impacts these protocols. Due to the pipelining, all tasks of the system execute with the same periodicity. .This results in nullifying the use of rate monotonic scheduling and converts normally dynamic scheduling protocols into static problems. Further the impact of tasks whose execution time is not fixed is addressed. Given knowledge of the minimum and maximum execution times of each task in the application's decomposed task set and the locking times of the shared communication resource, it is possible to establish a priority assignment for resolving conflicting requests for the shared resource. Further, given this scheduling and the task characteristics, it is possible to either establish the minimum cycle time (deadline) of the application which can be guaranteed or given a deadline requirement to predict whether the deadline can always be met.

While these results focused on a decomposition which resulted in a serial data flow, extension of these findings to systems where there exists parallelism in the data flow is anticipated. Also consideration of the merger of this work with work relating to automated or computer assisted decomposition of real-time applications is planned, the intent being to provide a tool for facilitating the implementation of hard real-time applications on multiprocessor systems.

REFERENCES

1. J. P. Lehoczky & L. Sha, "Performance of Real-Time Bus Scheduling Algorithms", Performance 86, 1986
2. C. L. Liu & J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Enviromnent" Journal of the ACM, Vol.20, No.1, Jan. 1982
3. A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", PhD Thesis, MIT, May 1983
4. R. Rajkumar, L. Sha, & J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors", IEEE Real Time Systems Symposium, Dec. 1988
5. L. Sha, R. Rajkumar, & J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol.39, No.9; Sep. 1990

Computer Music Performance as a Real-Time Testbed

David H. Jameson
IBM T.J. Watson Research Center
Yorktown Heights, New York
dhj@rhun.watson.ibm.com
January 14th 1991

Abstract

Traditionally, researchers in real-time think about sophisticated embedded systems such as controllers for nuclear power plants, aircraft and rockets, or robotics. Experimentation in these areas can be impractical, often expensive and potentially deadly. The domain of computer music offers an excellent and inexpensive testbed that exhibits real-time properties. Several features of the language ORE are used to simplify greatly the programming of a MIDI songfile player.

Introduction

Apart from soothing the savage beast, what else is music good for? Marvin Minsky has a wonderful analogy:

> *Each child spends endless days in curious ways; we call it "play". He plays with blocks and boxes, stacking them and packing them; he lines them up and knocks them down. What is that all about? Clearly, he is learning Space! But how on earth does one learn Time? Can one Time fit inside another, can two of Them go side by side? In Music, we find out!*
>
> [Minsky 82]

Of course, we have many more questions than the child but surely the principle is the same. In attempting to state what music really is, F. Richard Moore says "... that music addresses a sense for which our minds are the primary (if not the only) organ: our sense of *time*" [Moore 90].

Who among us has not at one time or another danced to the rhythm or sung along with the tune? Music is an art with which most people are intuitively comfortable. With time playing a major role in music it seems reasonable that given the tools to manipulate musical events, we can build systems to experiment with time-related issues and use our intuition and experience with music to help us.

MIDI

The introduction of MIDI (Musical Instrument Digital Interface) in the early 80s has led to a revolution in computer music. MIDI is a simple byte stream protocol that allows electronic musical instruments to communicate with each other. MIDI data is transmitted at a speed of 31Kb/s. For example, playing a note on one instrument could cause the corresponding note to be sounded on several other instruments. Thus the MIDI protocol

consists of control information. This information is divided up into commands such as NoteOn or NoteOff. These commands are examples of MIDI events.

It is important to remember that there is no explicit notion of time in the MIDI protocol to determine when notes should be played. In other words, when the sequencer sends a NoteOn (say) event out, it must be played immediately. The protocol does not support the concept of timestamped packets to be buffered by the receiving instrument for later playback.

If a computer is connected via MIDI, then it can record (store) incoming events and send those (possible transformed) events out again later. The best analogy to this process is the multi-track tape recorder except that instead of storing audio on each track, we only store events. Such a program is called a sequencer.

It is the responsibility of the sequencer to make sure that the appropriate data is sent to the correct synthesizer at the right time. Over the years, a variety of sequencers have been built for different computers and operating systems, each using a unique internal representation (of course!) of the data. Thus it was difficult to transfer music from one system to another except by connecting the systems together via MIDI, difficult to do if the systems are in different locations.

Therefore, a standard interchange format was devised and documented[1]. All modern sequencers can optionally store or load the music from a MIDI standard songfile. As part of the infrastructure for some experiments in interactive performance, I recently built a system that plays back standard songfiles. Using many of the concepts from ORE [DonnerJameson 86], several tricky issues were addressed and solved rather elegantly.

The MIDI songfile format

A songfile consists of a simple header followed by a set of tracks. The header specifies the number of tracks following and the clock resolution used when the file was created. The clock resolution is simply the number of ticks per quarter note. Typical values range from 120 t/q to 480 t/q. The number of ticks per second depends on the ticks per quarter note and the current temp. For example, if the current tempo is 120 bpm (beats per minute)[2] and the clock resolution is 480 t/q then the tick rate is 960 t/s, just over a millsecond per tick.

Tracks are of arbitrary length. Each track consists of a sequence of MIDI events, with each event being preceded by a delta. (Fig 1) The delta specifies, in ticks, how much time should pass before the following event occurs. Using ticks rather than an absolute time stamp makes it easier to apply some time transformations such as changing the tempo. If absolute time stamping were used, it would be necessary to modify the time stamp for every event whenever we wanted to change the speed of the music. By simply changing the speed at which ticks arrive, the tempo can be changed smoothly.

---

[1] The format is Standard MIDI File 1.0 and is specified by the International MIDI Association. An excellent description of this format is given by Steve De Furia [DeFuria et al 89]

[2] One beat = 1 quarter note. It's confusing that the "standard terminology" is ticks/quarter and not ticks per beat.

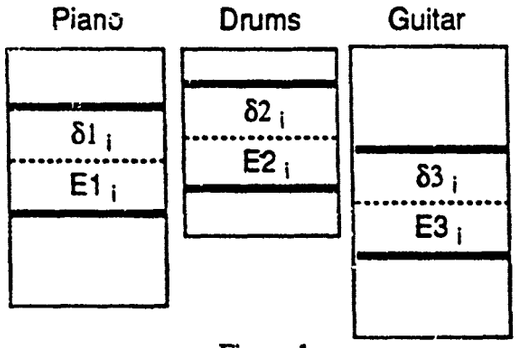


Figure 1
MIDI Songfile tracks

Playing songfiles

To play a songfile, the player must send the MIDI data in each track to the appropriate synthesizer at the correct time. To do this, the data of all tracks is typically merged into one buffer that is sorted by time of occurrence (measured in ticks).

The player discussed here is implemented using a different strategy. We take that the view that tracks should be treated as separate entities. A lightweight process (thread) is created to manage each track. There are two major reasons (other than aesthetics) why this view is taken. In the first place, it is much easier to manipulate (change, add or remove) specific track data when the tracks are distinct. This is very useful in case we wish to influence in real-time the performance of a piece of music, an important goal in this research. In the second place, keeping tracks separate provides for the opportunity to distribute the sequencer across multiple machines, allowing experiments with distributed real-time.

Associated then with each thread is a countdown timer. At the beginning, and after each MIDI event has been processed, the countdown is incremented by the delta preceding the next event. As ticks occur, the timer is decremented. When the timer reaches zero or goes negative, the thread wakes up and the next MIDI event is processed. Going negative means that the deadline has been missed. Incrementing the counter with the next delta rather than simply assigning the next delta assures that missed deadlines are not accumulated.

If processing a MIDI event just meant sending the bytes out, then the problem would be quite simple. We would just wait for the countdown to reach or pass 0 and then send the bytes out. Unfortunately there are two problems, one simple and one very subtle.

The simple problem is this - how do we know where the MIDI event starts and stops. There is no explicit length information available. For example, a NoteOn event consists of 3 bytes, 9x n v where n and v are the note number and velocity bytes respectively. x is a nibble specifying which one of 16 MIDI channels (0..15) is being used for the note. Some other events have a different number of bytes.

The second problem is the issue of "stuck" notes. For every NoteOn event sent, there must be a corresponding NoteOff event later on. If this is not so then some notes will just sound continuously. Normally the corresponding NoteOff event is in the songfile. But suppose we are playing a song and decide to stop in the middle. It is possible that we will stop

sending data before the NoteOff events have been seen causing notes to sound continuously[3].

To solve both of these problems, we must parse the songfile. Because we are playing the songfile directly, it must be parsed on the fly, as fast as possible. For this reason, I use a finite state machine that recognizes the complete MIDI grammar. The FSM also understands the delta and meta-events that occur only in songfiles.[4]

Transient information such as the current state is stored in a MIDI Control Block (MCB) one of which is created for each thread in the system that needs to parse MIDI.

Also defined in each MCB is an input function (IF) that specifies how to get the next byte. Further result functions (RF) exist that indicate what should be done when a MIDI event is recognized. Every one of these functions can be defined independently for each thread. An IF may get bytes from a track in a songfile or it may get bytes as they arrive from an external source if a musician is playing an instrument. The FSM uses the IF to read incoming data and the RFs are executed upon recognition of some event.

The default action for most RFs is to send the parsed event out to the MIDI interface. Since there is a separate RF for NoteOn and NoteOff events, it becomes trivial to define functions to keep track of these events. It is also possible to build functions to transform events in arbitrary ways.

In the next section we will see how this mechanism combined with ORE functionality allows for some sophisticated requirements to be implemented with ease.

Program structure

The core of the songfile player is a set of concurrent threads, one for each track, with each thread responsible for playing the music in its respective track. Using ORE notation and ignoring initialization of the MIDI control block variable mcb, the code for this is simple:

```
[
  < VAR mcb; GetByte(mcb); Transition(mcb); > -- Track 1
  ...
  < VAR mcb; GetByte(mcb); Transition(mcb); > -- Track n
]
```

Unfortunately, ORE has no mechanism for creating processes dynamically. Although there is some justification for this restriction on the grounds that it can be expensive to instantiate processes on the fly, if the number of processes needed can be determined during the initialization phase of the system, which is often the case, then a mechanism for dynamic creation of processes would be very convenient.

---

[3] Some keyboards come with a button called "panic" whose sole purpose is to send out all possible NoteOff events.

[4] Examples of meta events are commands to change the tempo or set the time signature. A discussion of these events is beyond the scope of this paper

The procedure GetByte is an input function (IF) that gets the next byte from a track. The procedure Transition updates the state of the finite state machine. A result function (RF) may be triggered as a consequence of the transition. So, for example, should the FSM recognize the variable length delta preceding each event, then the RF will cause the thread to sleep delta ticks.

In ORE, the threads above are siblings in the same lexical scope. Executing the program causes the songfile to be played in its entirety. However, we would like to add a little more control to this program. In particular, we would like the ability to stop the music at any time. In principle this is simple. We simply add the following sequence to the set of siblings above.

```
< Watch(KeyAvailable); Preempt; Break; >
```

This sequence sleeps until a key is pressed on the console. If a key is pressed, the thread will wake up, preempt all the siblings and then exit. Unfortunately, simply preempting threads is likely to cause the "stuck note" phenomenon described above. To solve this problem, we take advantage of the Last Will & Testament or LWT construct. The LWT allows us to define some code that should be executed if the sequence in which it is contained is subsequently preempted. Each thread responsible for playing a track now has the following structure.

```
<
  VAR mcb;
     LWT( "Send NoteOff for each unmatched NoteOn" )
     GetByte(mcb);
     Transition(mcb);
>
```

The RFs for NoteOn and NoteOff are extended so that they keep track of what notes are pending. Now, should the threads be preempted, each thread is able to die gracefully, making sure that it has left no stuck notes behind.

Discussion

The facility with which the songfile player was implemented shows the usefulness of ORE's semantics. The bookkeeping is trivial because it is not necessary for threads to have any global information, drastically simplifying the code and the ability to understand it.

Code was also added to display how well the system was meeting deadlines. although the system played music very well, it turned out that threads were often late by a tick or two. Depending on the clock resolution, this translated into several milliseconds , not enough to affect the output[5]. The missed deadlines occurred when notes in several tracks needed to be played simultaneously. There was then contention for the one shared resource in the system, the MIDI interface. Since it takes about 1ms to send the average MIDI event consisting of 3 bytes, it was easy to explain why deadlines were being missed.

It was rather gratifying however to find that the very simple scheduling scheme employed worked so well. Nevertheless, there is plenty of room to experiment with different

---

[5] Delays less than 8ms are undetectable by humans.

scheduling algorithms. Gross misbehavior in such algorithms will be quickly noticed due to the "strange" sound of the music that is heard[6].

Conclusion and future work

The songfile player is a simple but very useful application exhibiting deadline requirements. The equipment required was a PC and a synthesizer that cost under $400. The semantics found in ORE have been instrumental (no pun intended) in facilitating the development of the player.

Although the current system just plays "canned" songs, work is underway to add the ability to interact with it in various interesting ways. One interesting possibility is to attempt to model the interaction amongst the musicians in an orchestra and to influence (conduct) that orchestra. Although there have been several "conductor" projects, the focus seems to have been on controlling simple parameters such as global time and volume via novel user interfaces. [Mathews et al 80]

When several musicians play together, many new factors must be taken into consideration. For example, what do the cello players do when they notice (subconsciously?) that the violin section has speeded up slightly? Does the cello section increase to the same speed as the violin section or do they meet half-way? Do they perhaps get louder (or softer) as they change speed? How long does this response take? Some of this information is explicit in the score. Much of it requires interpretation and is influenced by the group. There are questions of synchronization of individual musicians and of feedback amongst them.

Another interesting issue involves using music notation concepts. Music notation is different from other languages in that time is built in. Since this notation has been working well for hundreds of years, one must ask if it is feasible to make a useful programming language out of it. One group [LoPall 89] has already suggested using music notation as an alternative for Gantt charts with time included. That paper defines the notation in terms of Real Time Logic [JahamianMok 86].Conversely, it might be reasonable to use music notation as a means of monitoring the real-time behavior of a system.

References

[DeFuria et al 89] "MIDI Programmer's Handbook"
Steve De Furia, Joe Scacciaferro, M & T Publishing, 1989

[DonnerJameson 88] "Language and Operating System Features for Real-time Programming"
Marc D. Donner, David H. Jameson, Computing Systems, Vol 1 No 1, Winter 1988

[JahamianMok 86] "Safety Analysis of Timing Properties in Real-Time Systems"
Farnam Jahamian & Al Mok, IEEE Transactions on Software Engineering, Vol 12 #9, September 1986

---

[6] Not to be confused with the strange music one often hears on the radio these days!

[LoPall 89] "RAGA: Musical Gantt Charts for Scheduling in Distributed Real Time Systems"
Virginia M. Lo, Gurdeep Singh Pall, Dept. Computer Science, University of Oregon, Unpublished?, October 20, 1989

[Mathews et al 80] "The Sequential Drum"
Max Mathews & Curtis Abbott, Computer Music Journal, Vol 4 #4, 1980

[Minsky 82] "Music, Mind and Meaning"
Marvin Minsky, Chapter 1, Music, Mind, and Brain: The Neuropsychology of Music, edited by Manfred Clynes, Plenum Press, 1982

[Moore 90] "Elements of Computer Music"
F. Richard Moore, Prentice Hall, 1990

SPECIFYING HARD REAL-TIME SOFTWARE: EXPERIENCE WITH A LANGUAGE AND A VERIFIER

Constance Heitmeyer and Bruce Labaw
Naval Research Laboratory
Washington, D.C. 20375

Introduction

Hard real-time (HRT) computer systems must deliver results within specified time intervals or face catastrophe. To detect timing problems in HRT systems, current development practice depends on exhaustive testing of the software code and extensive simulation. Unfortunately, this expensive and time-consuming process often fails to uncover subtle timing and other software errors. To improve this situation, research is needed in methods for specifying and verifying HRT systems.

At the Naval Research Laboratory (NRL), we advocate a three-phased approach to developing HRT systems that emphasizes the requirements phase of the software life-cycle. With this approach, 1) mathematically precise specifications of the timing and other system requirements are developed, 2) machine-based verification and other analysis tools are applied to the requirements specifications to improve their consistency and to insure compliance with critical timing and other properties, and 3) a semiautomated procedure is used to develop an implementation from the specifications. This implementation must meet the timing and functional constraints imposed by the requirements specifications.

NRL's current effort is focused on the first two phases of this approach. As part of this effort, we are building a software requirements toolset, containing tools developed at NRL as well as promising tools developed elsewhere. The toolset's goal is to help software developers specify, analyze, and verify the functional and timing requirements of HRT systems. Of special interest are tools that *scale up*, i.e., tools that are useful in specifying and verifying requirements of real-world, practical HRT software. Included in the NRL toolset are requirements generation tools and mechanical verifiers [Heitmeyer90]. The language supported by requirements generation tools should lead to formal, yet intuitive, specifications. Verifiers provide formal proof that given assertions about functional behavior and timing can be derived from the specifications; of special interest for real-time software are proofs that certain critical events occur within specified time intervals.

Although current commercial tools supporting HRT requirements specification are few and limited in capability, the SARTOR project at the University of Texas has developed two promising experimental tools. The first, a requirements generation tool, supports a graphical language, called Modechart [Jahanian88a, Jahanian91], that is designed to specify a system's timing requirements. The second, a verification tool, provides mechanical proof that a specification satisfies critical timing properties [Jahanian88b, Stuart90]. These prototype SARTOR tools are based on methods for specifying and analyzing timing properties that complement methods for specifying functional requirements [Heninger78, Heninger80] invented in NRL's Software Cost Reduction (SCR) project.

Recently, we developed Modechart specifications of several example systems and used SARTOR's prototype verifier [Stuart90] to prove the consistency of the specifications and selected timing assertions. Below, we present the Modechart specifications and timing assertions for one of these examples and summarize NRL's experience with both the Modechart language and the verifier, identifying their contributions to real-time software technology and recommending enhancements.

1. Example Modechart Specifications

This section introduces an example, provides Modechart specifications for the example, and presents two timing assertions that we proved about the specifications. The example, which is taken from the software requirements document of a real avionics system, the A-7E aircraft's Operational Flight Program (OFP) [Heninger78], has important features that make the specifications nontrivial, a shared resource (the display) and several different environmental inputs and outputs. The Modechart semantics used to construct the specifications are based on [Jahanian88a, Jahanian91]. The timing assertions are expressed in Real-Time Logic (RTL), a form of first-order logic invented by the SARTOR researchers to reason about time [Jahanian86].

Example: Pilot Data Entry and Display. Figure 1 provides Modechart specifications for a function performed by the OFP. The OFP software reads a character sequence (e.g., latitude or longitude) typed by the pilot and writes the sequence to a display panel. To initiate data entry, the pilot first presses the **DATA ENTRY** button. The software responds by turning on a keyboard light and clearing the display panel. Next, the pilot types a sequence of characters, which the software writes one character at a time to the display panel. Finally, the pilot presses the **ACCEPT** button to indicate that he has completed data entry. In response, the software turns off the keyboard light and clears the display panel.

To specify this example in Modechart, Figure 1 shows three top-level modes, called Pilot Input Recognizer, Data Entry and Display Function, and Output Generator. The Pilot Input Recognizer consists of three input drivers, one for each of the three hardware devices that the pilot uses to communicate with the software, namely, the **DATA ENTRY** button, the **ACCEPT** button, and the alphanumeric keyboard. The Data Entry and Display Function specifies how the system responds (i.e., what outputs it produces) to a sequence of inputs. The software response depends on both the event history as well as the input. In Modechart, a set of *modes* captures the event history, and *state variables* can be used to represent input and output data items. The Data Entry and Display Function receives input via changes to input data items and produces output by changing output data items. The Output Generator translates output data items into specific outputs (e.g., turn the keyboard light on). It consists of two drivers, one controlling the keyboard light, the other writing characters to the display panel.

Timing Assertions. Using the verifier, we proved the consistency of two timing assertions with the Modechart specifications. One assertion states that each data character is displayed within some fixed time interval after it is entered. Specifically, if t is the time that the pilot entered the ith character, then the time that the ith character appears on the display panel is at or before $t + 200$. To express this in RTL, we write

$$\forall i@((\text{HAVEDATUM} := T), i) \leq @((\text{DISPLAYED} := T), i) \wedge @((\text{DISPLAYED} := T), i) \leq @((\text{HAVEDATUM} := T), i) + 200. \quad (1)$$

Proving this assertion required the addition of two constraints to the Modechart specifications. First, we needed to bound the pilot's input rate. Given human performance limitations (humans can only type so fast), we assume a lower bound on the time interval between any two consecutive pilot key presses. Second, we needed to impose an order on the sequence of pilot inputs, since this assertion is only valid for certain pilot input sequences. The specifications in Figure 1 permit all possible pilot input sequences, even illegal sequences. The assertion in (1) is true only when the pilot enters a legal sequence: a **DATA ENTRY** followed by one or more data characters followed by an **ACCEPT**.[1]

A problem in proving (1) is that the ith entry into mode HAVE DATUM (i.e., pilot entry of the ith data character) does not correspond to the ith entry into mode DISPLAYED, because mode HAVE DATUM is only entered when the pilot enters a data character but mode DISPLAYED is entered when a data character is displayed *and* when one or more blanks are displayed. One solution is to replace the START DISPLAY mode in the display driver specification with two modes: START DISPLAY VAL (displays a data value) and START DISPLAY BLANK (displays one or more blanks). To express this in RTL, we write

$$\forall i@((\text{HAVEDATUM} := T), i) \leq @((\text{STARTDISPLAYVAL} := F), i) \wedge @((\text{STARTDISPLAYVAL} := F), i) \leq @((\text{HAVEDATUM} := T), i) + 200.$$

Because the verifier cannot prove formulas of this form, the assertion was rewritten in Modechart and proved with a *reachability* argument. To use such an argument, the Modechart specifications were augmented by adding an *unsafe* state, a state that violated the RTL assertion. Then, the verifier was executed on the augmented specifications to determine whether entry into the unsafe state was feasible. Because it was not, the assertion is considered proven. (We note that a solution that replaces the START DISPLAY mode with two new modes is undesirable for ease of change reasons. The output driver specifications should not be influenced by the requirements of the verification process.)

[1] In a complete specification of this example, the Data Entry and Display Function would also recognize illegal input sequences and generate appropriate responses (e.g., error messages).

The second timing assertion states that a minimum delay exists between the time that the last character of the character string is displayed and the time that a pilot press of the **ACCEPT** key is allowed. The rationale is that, before the pilot presses **ACCEPT**, he needs a minimum time to read and validate the string of characters that appear on the display. This assertion is expressed in RTL as[2]

$$\begin{aligned}\forall i \exists j @((\text{HAVEDATUM} := T), j) \leq @((\text{HAVEACCEPT} := T), i) \wedge \\ @((\text{HAVEACCEPT} := T), i) \leq @((\text{HAVEDATUM} := T), j+1) \wedge \\ @((\text{STARTDISPLAY} := F), 2i + j - 1) + 225 \leq @((\text{HAVEACCEPT} := T), i). \end{aligned} \quad (2)$$

To prove this assertion, we needed to augment the two constraints above with a third constraint that defines an upper bound on the character string length and that requires an **ACCEPT** to terminate each character string.

In developing the proofs of (1) and (2), an important consideration was whether the three constraints were requirements missing from the original specifications (shown in Figure 1) or whether they were simply logical statements needed to complete the verification process. We decided that two of the constraints should be added to the requirements specifications, in particular, the constraint describing human performance limitations and the constraint limiting character string length and requiring termination of a character string by an **ACCEPT**. How to specify these constraints (i.e., in Modechart, RTL, or some other format) and how to integrate them with the specifications in Figure 1 remains an issue, especially for the constraint describing human performance limitations. In contrast to the first two, the remaining constraint, which defines the legal pilot input sequences, was simply needed to complete the verification process; the statement we wished to prove concerns the system's response given legal pilot input. Hence, assertion (2) is incomplete: a complete statement of the assertion includes (2) as a consequent and a description of legal pilot input as an antecedent. How to specify this constraint is also an issue.

2. Contributions of Modechart and the SARTOR Verifier

Modechart. The SARTOR research effort has contributed to real-time software technology by providing an integrated approach to the specification and verification of critical timing properties. A crucial aspect of SARTOR is the Modechart language. While specifications in logic-based languages, such as RTL, other first-order languages [Heitmeyer83], and temporal logics like CTL [Clarke87] and RTTL [Ostroff89], facilitate machine-based analysis and verification, humans find such specifications hard to produce and hard to understand (e.g., see [Jaffe89]). In contrast, we found the graphical Modechart specifications highly readable and relatively easy to generate. Although complete graphical specifications of the requirements may be impractical for large systems, the readability of the Modechart specifications make them very useful during the process of constructing the requirements specifications.

A fundamental contribution of Modechart is the ease with which specifiers can use the language to understand and reason about a system's timing behavior. Specifiers can first use modes, actions, events, and state variables to define the parallelism and sequential behavior inherent in the application domain. We found that Modechart's hierarchical structure facilitated the construction of our specifications by allowing us to combine top-down and bottom-up approaches. Once the system's *functional behavior* (sometimes called control structure) is defined, then timing behavior can easily be added in terms of deadlines and delays.

Unlike temporal logics, such as CTL and RTTL, which are designed to specify the temporal ordering of events, Modechart and RTL are designed to specify both the temporal ordering of events and the temporal distance between events. In real-time systems, constraints on the temporal ordering of events are insufficient. In such systems, certain critical events (e.g., the firing of a weapon, an alert signaling the spill of a hazardous substance) need to occur within specified time intervals. Unlike languages based on temporal logic, Modechart and RTL provide a compact notation for defining the timing constraints imposed on critical events. These constraints are described in Modechart by delays and deadlines, in RTL by the occurrence function.

In addition to producing highly readable specifications that compactly express both temporal ordering and temporal distance, Modechart has additional benefits lacking in other specification languages. Unlike

[2] The second clause of the formula is only checked if $@((\text{HAVEDATUM} := T), j+1)$ is defined, that is, if the pilot has entered a character following the **ACCEPT**.

[Heninger78], which describes only the software requirements (represented in our specifications by the Data Entry and Display Function), Modechart can describe the complete *system requirements*. The inclusion in Modechart of external events as well as action completions makes the specification of the complete system requirements possible. An additional benefit is Modechart's support for concurrency. In a HRT system, input devices, output devices, and computers need to operate concurrently, and their behavior needs to be synchronized. The parallel modes included in Modechart make the description of concurrency possible, while Modechart's state variables enable synchronization and communication among parallel modes. A third benefit is Modechart's support of nondeterminism. As Gabrielian has noted [Gabrielian90], in some parts of a specification, an event or condition may trigger a transition from one mode to more than one other mode. If the actual requirements permit all possible transitions, forcing a transition to exactly one mode is a premature design decision. Modechart's semantics allow the specifications to express such nondeterminism.

Prototype Verifier. We found the prototype verifier useful in improving the correctness and the completeness of our sample specifications. While human proofs of timing assertions are feasible, such proofs often contain errors, first, because proofs involving inequalities and substitutions are tedious, and, second, because humans may fail to provide complete proofs, especially for boundary cases. The specifier not only allowed us to detect such errors but also increased our overall understanding of the specifications, especially the interactions of individual components. However, while helpful, verification tools do not free the human from thinking about the logic of the specifications They provide mechanical assistance for checking the logic. Our experience suggests that humans working with a mechanical verifier are more likely to find errors in the specifications' logic than humans doing manual verification alone.

3. Limitations of Modechart and the Verifier

As noted earlier, the SARTOR tools are prototypes. Below, we identify problems that emerged when we applied the Modechart language and the SARTOR verifier to our examples and suggest ways in which the tools could be more fully developed. For a full discussion and examples of these and some additional problems, see [Heitmeyer91]. One general comment about both Modechart and RTL concerns expressiveness. In some cases, a constraint was more easily expressed in one language than the other. For example, a Modechart specification of the legal pilot input sequences is straightforward, whereas a specification of the sequences in first-order logic is tedious and less intuitive, requiring considerable notation for bookkeeping purposes. In contrast, given a set of Modechart specifications, defining timing constraints involving nonadjacent modes (see below) is easier in RTL than in Modechart. Further analysis is needed to identify other classes of constraints that are more easily expressed in one language than in the other.

Modechart Limitations.

No Support for Shared Resources. When a resource, such as a display device, is shared by more than a single action, relating the ith occurrences of two events, one involving the shared device, may be impossible in Modechart.

Restricted Modechart Functionality. In generating sample Modechart specifications, we identified some cases in which Modechart's functionality was overly restricted. One example of limited functionality concerns self-looping: Modechart currently prohibits a transition from a mode to itself.

Inability to Define Timing Constraints on Non-Adjacent Modes. By non-adjacent modes, we mean two modes between which no mode transition exists. In Modechart, it is impossible to express timing constraints involving non-adjacent modes without creating dummy modes. (We regard dummy modes as undesirable artifacts that add to the specifications' complexity.)

Verifier Limitations.

Lack of User-Friendly Feedback. As [Rushby89] has stated, determining whether an assertion is valid is only one of the useful functions that a verifier performs. In addition, a verifier should support an interactive human-computer dialogue that enhances human understanding of the specifications and that facilitates reasoning about them. To date, little attention has yet been paid to the SARTOR verifier's user interface. Although a high-quality user interface was not a goal of the effort producing the current SARTOR verifier, a better user interface is needed before the tool can be used in a production environment.

Inability to Determine Bounds on Timing Variables. In some cases, specifiers may have estimates of some timing constraints. In such cases, it should be possible to represent unknown processor times (e.g.,

the time needed by the computer to perform a particular computation) with variables and to use the known timing information and global timing assertions to derive bounds on these variables. Future versions of the verifier should derive bounds on timing variables.

Limited Formula Repertoire. At present, the verifier only supports a small number of RTL formulas (see [Stuart90]). To prove the assertions presented in Section 1, we needed several additional formulas (e.g., $A \rightarrow B$, where A and B are logical statements). A version of the verifier is needed that proves such formulas in their original form.

Inflexible Input Form. In some cases, we wished to prove an assertion about a combination of Modechart specifications and one or more RTL assertions. Unfortunately, the current verifier can only prove assertions about Modechart specifications. Enhancing the verifier to prove properties about a combination of Modechart specifications and RTL assertions would be useful.

4. Summary

In our view, Modechart and the SARTOR verifier represent a significant advance in the state-of-the-art of specification and verification of HRT systems. A major advantage of Modechart specifications is their readability and the ease with which specifiers can use the language to reason about timing. Moreover, unlike temporal logics, Modechart specifications can compactly express both temporal distance and temporal ordering. The SARTOR verifier demonstrates the feasibility of machine-based proofs that Modechart specifications have selected timing properties.

A final comment concerns the scaleability of Modechart. Little is known about the utility of Modechart, RTL, and the SARTOR verifier for building real-world systems. Our experiments suggest that Modechart's scaleability is limited: specifying large quantities of requirements data in graphical form is probably impractical. But this doesn't mean that Modechart isn't useful. In our view, more than a single approach to real-time requirements specification is needed. Because it produces highly readable, intuitive specifications, Modechart may be most appropriate during the process of building the requirements specification. In contrast, the tabular formats for requirements specification introduced in SCR (see [Heninger78, vanSchouwen90] for examples) are more appropriate in a reference document. These formats provide the reader with less intuition about the requirements than the graphical notation but do concisely and formally describe the large volume of requirements data associated with real-world, practical software. For these reasons, the toolset we are constructing supports both the Modechart and the SCR 'views' of the requirements data. Our future goal is to develop a single conceptual model that supports both 'views'.

Acknowledgments. We are especially grateful to Paul Clements of the University of Texas and NRL and John Gannon of the University of Maryland for many valuable discussions. We also thank Al Mok and Doug Stuart of the University of Texas for allowing us to experiment with the SARTOR tools and for their openness to our suggestions concerning extensions. Finally, we thank the other members of our project team, Carolyn Brophy and Anne Rose.

REFERENCES

[Clarke87] E.M. Clarke and O. Grumberg, "Research on Automatic Verification of Finite-State Concurrent Systems," *Ann. Rev. Comput. Sci. 2*, 269-90, 1987.

[Gabrielian90] A. Gabrielian et al., "Specifying Real-Time Systems with *Extended* Hierarchical Multi-State (HMS) Machines," Thomson-CSF, Inc., report 90-21, Jan. 1990.

[Heitmeyer83] C. Heitmeyer and J. McLean, "Abstract Requirements Specifications: A New Approach and Its Application, *IEEE Trans. Softw. Eng. SE-9*, 5, Sep. 1983, 580-589.

[Heitmeyer90] C. Heitmeyer and B. Labaw, "Software Development for Hard Real-Time Systems," *Proceedings, Seventh IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, VA, 10-11 May 1990.

[Heitmeyer91] C. Heitmeyer and B. Labaw, "Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier," NRL report (in press).

[Heninger78] K.L. Heninger et al., "Software Requirements for the A-7E Aircraft," NRL Rep. 3876, Nov., 1978.

[Heninger80] K.L. Heninger, "Specifying Software Requirements for Complex systems: New Techniques and Their Application," *IEEE Trans. Softw. Eng. SE-6*, 1, Jan. 1980.

[Jaffe89] M.S. Jaffe and N.G. Leveson, "Completeness, Robustness, and Safety in Real-Time Software Requirements," Univ. of Calif., Irvine, TR 89-01.

[Jahanian86] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Softw. Eng. SE-12*, 9, Sep. 1986, 890-904.

[Jahanian88a] F. Jahanian et al., "Semantics of MODECHART in Real Time Logic," *Proceedings, 21st Hawaii Intern. Conf. on System Sciences*, Jan. 5-8, 1988.

[Jahanian88b] F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of MODECHART Specifications," *Proceedings, Real-Time Systems Symposium*, Huntsville, AL, Dec., 1988.

[Jahanian91] F. Jahanian and A. K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Softw. Eng.* (in press).

[Ostroff89] J.S. Ostroff, "Real-Time Temporal Logic Decision Procedures," *Proceedings, Real-Time Systems Symposium*, Santa Monica, CA, Dec. 5-7, 1989, 92-101.

[Rushby89] J. Rushby and F. von Henke, "Formal Verification of the Interactive Convergence Clock Synchronization Algorithm using EHDM," SRI-CSL 89-3, SRI International, Menlo Park, CA, Feb. 1989.

[Stuart90] D.A. Stuart, "Implementing a Verifier for Real-Time Systems," *Proceedings, Real-Time Systems Symposium*, Orlando, FL, Dec. 5-7, 1990, 62-71.

[vanSchouwen90] A.J. van Schouwen, "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application for Monitoring Systems," Queen's Univ., Kingston, Ontario, TR 90-276, May 1990.

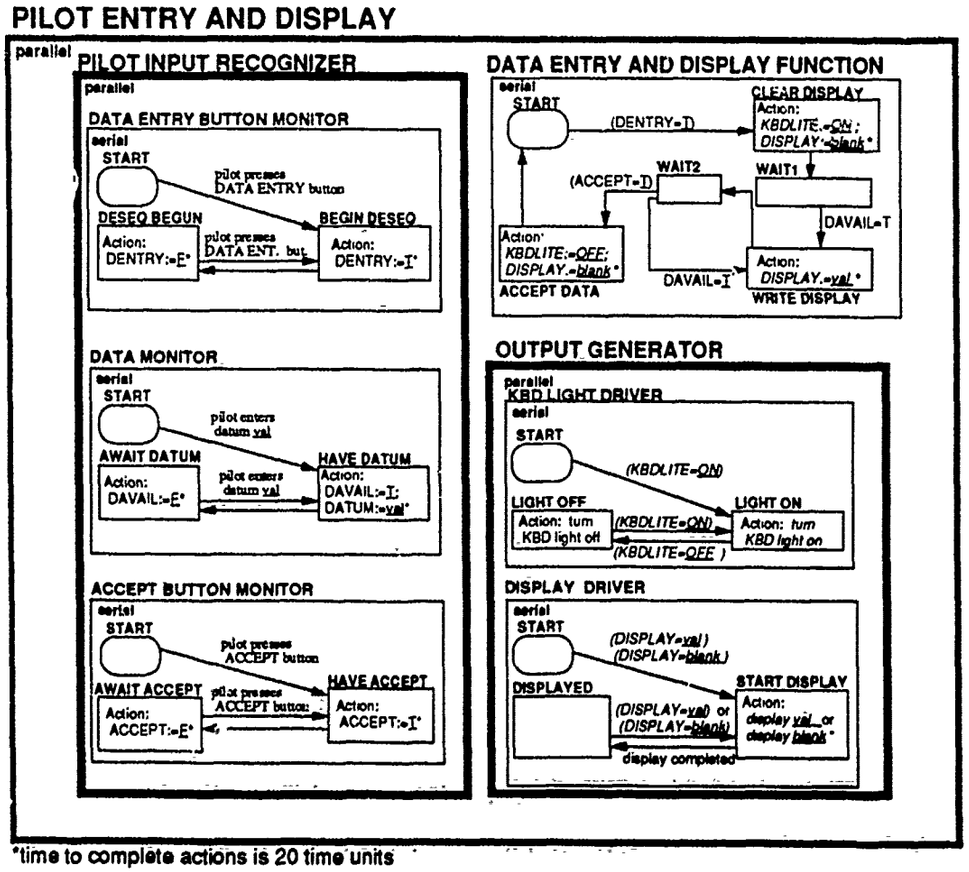


Figure 1. Modechart Specification of Pilot Entry and Display Example

Designing a Hard Real-Time System with Automatic Memory Management

Edward E. Ferguson, Dexter S. Cook, and David H. Bartley
Computer Systems Laboratory
Computer Science Center
Texas Instruments Incorporated
Dallas, Texas

January 15, 1991

1 Automatic Memory Management

Run-time support systems for most modern programming languages provide a memory management package with which the user can dynamically allocate and deallocate storage for objects. Some memory management packages are automatic in the sense that they reclaim the storage associated with an object when the package can determine that the application can no longer use that object. The component that performs this service is commonly called the garbage collector.

Automatic memory management is a valuable tool for complex applications. It can simplify the task of an application programmer since he can use complex dynamic data structures without having to explicitly determine constituent lifetimes. This freedom can result in more robust applications since subtle errors involving premature or missed deallocation are not possible [4]. Automatic memory management is fundamental to LISP-based languages and is desirable in languages such as Ada, C, and C++.

Many garbage collection algorithms have been described in the literature (e.g., [7,2,5,1]) that are real-time in the sense that they can be guaranteed to reclaim memory faster than the application can allocate it. This macro-level property is not sufficient for a hard real-time system since it is still possible for deadlines to be missed due to micro-level side effects of executing the collector. We have developed a new reclamation algorithm that is compatible with hard real-time execution.

2 Our Project

Our group has prepared an experimental programming system for embedded applications which require both hard real-time scheduling and automatic memory management. The system includes tools with which to compute or estimate an upper bound on the execution time of a task and to evaluate the worst-case scheduling behavior of an application based on the timing parameters of its tasks. The run-time system supports several languages, has an operating system kernel for

stand-alone execution, and interleaves execution of multiple tasks with a priority-based preemptive scheduler. The system is general-purpose in the sense that it is intended to be used for a number of types of embedded applications with both hard and soft real-time components. This requires a general-purpose automatic memory management package, not one that can be tailored to the special characteristics of one class of application. For example, it is not acceptable to achieve a bound on the execution time of the collector by demanding that an application not generate objects larger than some specified small size.

3 Design Problems

We encountered three primary problem areas when we added automatic memory management to a hard real-time system:

- Providing adequate computational resources to reclaim storage.
  The time required to reclaim memory is typically much longer than the execution times (and deadlines) of the hard real-time tasks since reclamation must access a significant fraction of system memory. For the system to be responsive to external events while reclamation is occurring, an algorithm must be designed so its execution can be interleaved with application tasks.

- Minimizing the impact of the collector on the schedulability of application tasks.
  Reclamation algorithms typically involve scanning objects, relocating pointers, and copying objects. These operations cannot be safely interrupted by application tasks that use the automatic memory management abstraction and thus become critical sections during which execution of the application is blocked. It is essential that the durations of such atomic operations be tightly bounded so the reclamation process's interference with the application tasks can be assessed and held to practical values. For example, an algorithm that must copy an object atomically is undesirable if worst-case behavior must be handled: any task has the potential to be delayed for the time required to copy the largest allocated object.

- Maintaining predictable execution times for application tasks.
  The choice of reclamation algorithm can affect the execution time of an application task due the overhead associated with invoking the memory management abstraction. For example, some designs (e.g., [2]) potentially require that an application task pause to copy an object on any access. It is important to design an algorithm that has a uniform and tightly bound overhead so it is possible to automatically or manually determine the execution time of application tasks.

4 Our Design

We designed and implemented an automatic memory management abstraction [6] that solves the problems enumerated in the previous section:

- Providing adequate computational resources to reclaim storage.
  The garbage collector is implemented as a separate task that executes concurrently with the application and can be preempted by time-critical tasks. Executing the collector as a

separate task means the rate at which storage is reclaimed can be adjusted via standard scheduling parameters of the collector. Thus any techniques that are developed to make dynamic changes to the characteristics of application tasks can also be employed to tune the behavior of the collector.

- Minimizing the impact of the collector on the schedulability of application tasks.
  Each critical section in the collector has a bounded duration that is independent of object size so the worst-case blockage of application tasks can be determined. For example, if a scheduling policy based on [3] is used, then an application task with higher priority than the collector can be delayed by at most the duration of the longest single critical section of the collector.

- Maintaining predictable execution times for application tasks.
  The abstraction provides the following operations to application tasks:

  - Create an object
  - Read a field in an object
  - Write a field in an object
  - Determine if two pointers refer to the same object

  The critical sections in these operations execute for bounded intervals that are independent of object size.

Since the application continues to allocate storage while the collector executes, the choice of the collector's scheduling parameters is subject to a number of tradeoffs. For example, assume that a reclamation algorithm is used that copies usable objects to implicitly find unusable ones (e.g., [1]). Since the collector executes concurrently with the application, collection must begin early enough to ensure that there is enough free space to hold both the storage allocated concurrently by the application and the storage copied by the collector. If the collector is given a low percentage of the processor time, the net amount of memory available to the application is reduced because collection must begin relatively early to compensate for the relatively long elapsed time required for the collector to finish. If the collector is given a high percentage of the processor time, memory is used more efficiently but the reclamation process may degrade the response time of the application. Typical parameters required to configure the collector are the size of managed memory, the rate at which the application allocates storage, the maximum percentage of allocated storage that will contain usable objects, and the internal time constants of the collector.

5 Conclusions

We have briefly described some of the design issues that arose while building a development system for embedded applications that integrates automatic memory management with hard real-time scheduling. We have implemented an automatic memory management abstraction with the property that each associated critical section has a fixed execution bound that is independent of object size [6]. Reclamation is performed in a standard task that executes concurrently with the application. This system currently executes on embedded computers based on the MIPS R3000 processor. We are now characterizing the behavior of the collector so it can be made adaptive via dynamic changes to its scheduling parameters.

References

[1] Andrew W. Appel, John R. Ellis, and Ki Li. "Real-Time Concurrent Collection on Stock Multiprocessors," *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 11-20, 1988.

[2] Henry G. Baker, Jr. "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, 21(4):280-294, April 1978.

[3] T. P. Baker. "A Stack-Based Resource Allocation Policy for Realtime Processes," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 191-200, 1990.

[4] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment," *Software - Practice and Experience*, 18(9), 807-820, September 1988.

[5] Rodney A. Brooks. "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware," *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 256-262, 1984.

[6] Edward E. Ferguson, Dexter S. Cook, and David H. Bartley. *To appear.*

[7] Philip L. Wadler. "Analysis of an Algorithm for Real Time Garbage Collection," *Communications of the ACM*, 19(9), 491-500, September 1976.

Application of Partial Evaluation to Hard Real-Time Programming

Vivek Nirkhe* William Pugh†

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Many real-time programming languages restrict the use of high-level programming features since their use makes it difficult to determine the maximum execution time of program at compile-time. This makes it difficult to write truly *reusable* real-time programs. In this paper, we claim that the technique of *partial evaluation* provides a solution to this problem. Partial evaluation allows us to use information about the execution environment to create specialized versions of general programs. A program specialized for a particular environment displays the same behavior as the original program and has predictable execution time.

1 Introduction

This paper deals with programming language support for *hard real-time* systems. In these systems, failure to meet deadlines of the application is fatal and thus has to be avoided. Predictable timing behavior of these applications can be ensured using scheduling algorithms that use the application deadlines and the execution times to make scheduling decisions. The key requirement of these algorithms is the complete and *deterministic* knowledge of timing properties of the applications before run-time. The traditional stochastic information such as average execution time are not appropriate for hard real-time systems. In some instances the time needed for different parts (or blocks) of the code and precedence relations between blocks are also required for proper scheduling.

Hence, a real-time programming programming language has to facilitate both the expression of the time constraints of an application and the ability to predict its execution time or an upper bound thereof. Some examples of the time constraints of real-time application are the earliest start-time, the latest end-time, and the periodicity. The execution time of an application program depends on factors such as the algorithm, the programming language, and the hardware. The ability to find the execution time of an application program is complicated by many language features. These include conditional statements, loops, recursion, use of files, dynamic data structures, and non-determinism. These features make it impossible to predict in sufficient detail the execution path the application will take at run-time to obtain a tight upper bound on the execution time. A naive solution to this problem is to restrict the use of these features or forbid them altogether. However, such a restriction forces the system designers to develop programs dedicated to each new environment. It is not possible to write general programs, which are not tied to a particular environment and reuse them without any changes to the source code.

Our approach to this problem is based on the technique of *partial evaluation*. In this technique, partial data (partial information available about the execution environment) is used to derive a specialized version of a general program. For a real-time program, we use this technique to determine the branches of conditionals the program would take, and to unwind the loops and recursion where necessary to meet the time constraints. It also allows us to determine the maximum loop counts. The new program is devoid of the conditional statements, loops, or recursion which affect the prediction of the execution time. ¿From a given program, partial evaluation produces a new program where the execution path can be obtained

*Supported in part by contract DSAG60-87-C-0066 from the U.S. Army Strategic Defense Command.
†Supported by NSF grant CCR-8908900.

accurately. The new also program has the same behavior as the original for the given data.

In order to predict the execution time, it is necessary that the bounds of loops and recursion are known. Hence, our technique allows the use of these features such that their bounds can be determined from the source code at compile-time itself. Parameters of the target environment that affect the execution path and its length are distinguished in the programs and hence can be extracted easily. It forbids the loops and recursion whose bounds do not depend upon the execution environment making it impossible for the compiler to predict the execution time. In our language, we provide a type distinction to distinguish the variables which affect the execution path and hence must be known at compile-time and the ones which do not. Thus, knowing the environment, the execution time can be accurately predicted. Using this technique, real-time programmers can develop truly reusable programs employing high level control structures while still meeting the requirements of hard real-time operating systems.

1.1 Related Work

Most research in this area has been directed towards finding the maximum execution time of real-time programs. The group at the University of Texas at Austin has designed a timing tool that uses annotations of the source programs to find the worst case path in the assembly language programs produced by a compiler[Mea89]. In the MARS project[PK89], a tree, similar to the syntax tree, of the timing information of the program is created and the time of the longest path is obtained. This tree can be edited interactively by the designer to improve the timing estimates. In the Real-Time Euclid[KS86] project, the aim is also to find the worst case execution time. Language constructs such as loops and semaphores are augmented with the addition of time bounds that they must adhere to. The language forbids recursion and dynamic data structures. Park and Shaw[PS90] are concerned with finding the lower and upper bounds on the execution time and base their work on the concept of timing schemas. They take into account both machine independent and machine dependent parts of the code produced by a compiler. The current version of the tool requires programmer intervention for timing decisions of loops and does not permit recursion.

We feel that the worst case estimate is not an appropriate measure as it can involve waste of resources in most instances. Hence, we want to estimate the execution time for each execution environment separately. In addition, we need to know the execution time required for each block separately to facilitate pre-scheduling. Apart from these goals, we wish to remove the language restrictions such as disallowing recursion and putting fixed upper bounds on the loop counts as we feel that they are unnatural restrictions.

2 Partial Evaluation

The central concept of *partial evaluation* is to create, from a general purpose program, a version of a program specialized for a specific environment[Ers82]. A partial evaluator uses partial data to evaluate portions of the program in advance and create a new program customized for the specific environment.

The following simple (but contrived) example illustrates the above process. Consider the following program fragment, which finds the value of (a + b)^n:

```
for i = 1 to n do
   if (i == 1)
      p = a + b;
   else
      p = p * (a + b);
```

A partial evaluator will specialize this program for an environment where it is known that n is 2 and b is 0, producing the folowing program fragment to find the square of a :

```
p = a;
p = p * a;
```

While the example given above is trivial, partial evaluation has been applied to interpreters to obtain compilers[JSS85]. Partial evaluation has also been advocated for situations where a general-purpose program must be run very efficiently on a computationally expensive problem. Often, enough information about the environment is available at compile-time to allow the partial evaluator to perform advance data manipulation. This approach reduces or eliminates the programmer's high level data and control abstraction to obtain efficient lower-level programs[Ber90, GMT85].

3 Partial Evaluation in the Real-Time Domain

We look to partial evaluation to solve a number of problems in hard real-time programming, arising from the fact that we must be able to pre-schedule the execution. We need a tight compile-time upper bound on the execution time of any block of code. This would seem to prohibit the development of reusable programs. By reusable programs, we mean the same source code that can be translated into different executable versions for different applications (where they will take different amounts of time). This same restriction also seems to prohibit recursion and first-class functions i.e. the ones that can be stored in tables, copied and executed.

For example, consider an interrupt polling module, that takes as an input a table of interrupts to be polled and actions to be taken on them. A generic polling module can not be constructed, since while compiling the generic module, we would not know an upper bound on the time taken for a round of polling, as it depends on the size of the table and each of the individual actions. In an application, the interrupt polling module might be used from several places. Each use would call the polling module with a specific interrupt table known at compile time. By using partial evaluation to automatically produce a specialized version of the generic polling module for each use, we can still perform hard real-time scheduling of the program. An error will be reported if the interrupt polling module is called with arguments that can not be adequately determined at compile-time. This technique can also be applied to a variety of problems such as a generic assembly line controller, or a recursive merge sort routine which are discussed below in brief.

We believe that this approach would allow real-time programmers to write programs that are not dedicated to one target environment. They can use the high-level, reuse-oriented programming styles as are used by other programmers. We contend that, the assumption that the partial data is available at compile-time is a valid assumption for hard real-time systems. Due to the nature of these systems, the system designers know the environment of the target system in detail.

This approach requires that the programs should not involve unbounded loops or recursion that cannot be bounded at compile-time. This is not an undue restriction as the real-time programs must have bounded execution times to meet the application deadlines. Moreover, it promotes a useful programming discipline.

3.1 Maruti Partial Evaluation

Maruti is a testbed for research in hard real-time systems and is based on the technique of pre-scheduling[LA90]. In this technique, the application execution is guaranteed by reserving the required resources prior to run-time. This takes into account the time constraints, time requirements, and dependencies due to communication and synchronisation. Pre-scheduling does away with run-time contention for resources thus reducing the unpredictability.

The knowledge of the maximum execution time alone is not sufficient for pre-scheduling. In addition, pre-scheduling requires separate execution times of blocks and the temporal relations between them. The temporal relations between blocks can be obtained using the time constraints of each block and the precedence between them. MPL (Maruti Programming Language)[NTA90], an object-oriented language, provides support for this by allowing expression of explicit timing constraints. In Maruti, the compiler (with the help of related tools such as partial evaluator) produces a *computation graph* which depicts the temporal and precedence relations between blocks. The edges are annotated with timing constraints and the nodes with execution time requirements. The scheduler uses this information to derive a schedule and verify that the constraints are met.

In Maruti, the first phase of compilation is partial evaluation which performs source-to-source transformation of programs. It also produces a computation graph of the program. The original programs may incorporate unbounded while loops, recursion, first-class functions, and generic modules. If no errors are found during partial evaluation, a new program will be produced in a restricted form where all loops have known maximum upper bounds, the depth of the recursive procedures is known and the execution time of all blocks is determinable. The known techniques of predicting execution time can then be used for such programs. The resulting computation graph describes the time constraints and precedence relations, such as message passing, remote procedure call, and synchronization between blocks. A block consists of statements that have the same temporal scope (i.e. the time between the earliest start-time and the latest end-time). The MPL temporal constructs[NTA90] define the temporal scope of statements, whereas the

control flow statements, communication primitives, and concurrency and synchronisation primitives define precedence relations. Blocks denote the units which require separate scheduling decisions.

In order to facilitate partial evaluation, we define two kinds of variables in the language, namely, *compile-time variables* and *run-time variables*. The restriction on compile-time variables is that their values have to be known at every step of partial evaluation (with some exceptions, see below). Hence, it is a type error to assign a run-time value to a compile-time variable. The remaining variables, similar to the usual variables, are called run-time variables. We also define an expression to be compile-time if it does not refer to a run-time variable, which is a type restriction.

The above distinction provides us with simple type inferencing and easy partial evaluation without the combinatorial explosion of the state of the partial evaluator. In order to ensure that the upper bounds of the loops (and depth of the recursion) are known at compile-time, we require that they are represented using compile-time variables. Use of loops and recursion whose bounds are run-time variables is a type error. Similarly, if the index of the loop goes beyond the known maximum upper bound an error is detected. Thus the above distinction is useful for good error reporting during partial evaluation.

Partial Evaluation of Statements

Here we consider the restrictions placed by partial evaluation on each control flow statement and describe the resulting program. For brevity, we do not go into details; they can be found in [NP90].

During partial evaluation, each assignment statement either results in a new value being assigned to a compile-time variable or results in a new assignment statement for the residual program. This depends upon the types of the two sides of the assignment statement.

For a conditional statement, we require that the condition must be a compile-time expression, if the conditional statement encloses a statement requiring separate scheduling. Depending upon the value of the condition, a conditional statement, is replaced by one of its branches. If the condition is a run-time variable, a new conditional statement is created in the new program.

For the the loop statement, the lower and upper limits of the loop index must be compile-time expressions if any statement in the loop body requires separate scheduling. In this case the loop gets unrolled. If the limits are not compile-time expressions, the maximum number of loop iterations must be specified to be able to predict the execution time of the block. Other loop statements are handled similarly, except that the partial evaluation may not terminate for the **while** and **repeat** ...**until** loops. A possible error can be signaled after a sufficiently large number of loop body evaluations.

Given the definitions of procedures and calls to them with some compile-time variables as parameters, we specialize the definitions for each call separately. For each call, the body of the procedure is then further partially evaluated. Even the recursive procedures can be handled in this manner. However, it is possible that the partial evaluator may never terminate while evaluating recursive procedures. In such a case also, a possible error can be signaled after a large number of evaluations of recursive calls. Partial evaluation of the temporal statements is similar to other control flow statement except that they are used to create and annotate the computation graph with timing constraints.

3.2 Examples

The following example illustrates the application of partial evaluation to a generic polling module, mentioned above. The polling module uses a table that lists the periodicities of polling each interrupt, interrupt flag routines, and the actions to be taken. The specified action is taken only if the interrupt is ready i.e. the function, which returns the interrupt flag, evaluates to true.

```
struct {
  int     period;
  boolean (*ready)();
  void    (*action)();
} intpt;

void
  poll (compiletime int N; intpt *table)
{
  compiletime var
    int cm, small, large, i;
  runtime var
    int dev, cycle;

  /* find the common multiple of the periods */
  cm = 1;
  for i = 1 to N do {
```

```
    small = min (cm, table[i].period);
    large = max (cm, table[i].period);
    if (large mod small == 0) then
      cm = large
    else
      cm = small * large;
  }

  /* poll all devices during that period */
  for cycle = 1 to cm do
    within onecycle do
      for dev = 1 to N do
        if ((cycle % table[dev].period)==0) then
          if ((*table[dev].ready)()) then
            (*table[dev].action)();
}
...
compiletime var
  int tot_dev = 3;
  intpt loc_tab[3] = {{2, &ready1, &action1},
                      {1, &ready2, &action2},
                      {3, &ready3, &action3}};
...
from starttime to endtime every bigperiod do
  (void) poll (tot_dev, &loc_tab);
```

Given the partial information about the periodicity, the interrupt flag routines, and the interrupt handlers, the above code can be partially evaluated. The code for polling module is produced assuming the inline expansion of procedures, though, we have a choice between an inline expansion and an actual call at runtime to a modified procedure. For the lack of space, routines ready and action are not expanded.

```
from starttime to endtime every bigperiod do {
  within onecycle do {
    if (ready2()) then action2();
  }
  within onecycle do {
    if (ready1()) then action1();
    if (ready2()) then action2();
  }
  within onecycle do {
    if (ready2()) then action2();
    if (ready3()) then action3();
  }
  within onecycle do {
    if (ready1()) then action1();
    if (ready2()) then action2();
  }
  within onecycle do {
    if (ready2()) then action2();
  }
  within onecycle do {
    if (ready1()) then action1();
    if (ready2()) then action2();
    if (ready3()) then action3();
  }
}
```

This technique can also be applied to recursion e.g. a recursive merge-sort routine where the dimension of the array is known. The partial evaluator will unwind the recursion prior to run-time to create a non-recursive code to perform the sort. The resulting code can be analyzed to get a tight upper bound on the execution time.

Another application of this technique would be a generic assembly line controller. Such a module can be used in different settings provided the number of stations, time to service in each station and the distance between them. At the same time, the turn around time and other parameters can be predicted.

3.3 Work in progress

We have defined the partial evaluator for a subset of the MPL in terms of denotational semantics[NP90]. The partial evaluator provides two outputs, namely, the computation graph and the residual program. The nodes of the computation graph denote blocks of the residual program, whereas, the edges denote precedence relations. This graph is augmented with the timing constraints of the blocks.

We have also defined the denotational semantics for the MPL. We have been able to prove the correctness of the partial evaluator. In brief, it can be proved that for a given program p, $\mathcal{C}[\![p]\!] = \mathcal{C}[\![\mathcal{C}_{partial}[\![p]\!]\,]\!]$, where $\mathcal{C}$ and $\mathcal{C}_{partial}$ are the semantics of the interpreter and the partial evaluator respectively. This states that the semantics of a given program is the same irrespective of whether it is interpreted directly or first evaluated partially and then the residual program is interpreted. This shows that the partially evaluated program behaves in the same manner as the original program and the semantics of the program is not changed.

4 Future Work

We are currently working on extending this approach to concurrency and synchronization primitives of the MPL. The difficulty in applying partial evaluation to these statements lies in predicting the order of execution between concurrent statements that share read-write variables. If the order chosen for partial evalua-

tion is different from that chosen by the scheduler, the results of the partial evaluation will be wrong. We are also trying to apply this technique to exception handling and interrupt processing; some work has already been done in this direction. This approach can be extended easily to obtain the memory requirement of a process. This is possible since we know the bounds on all loops in the program after partial evaluation. Future work will consist of implementing the partial evaluator and the compiler of the language and then interfacing them with the remaining Maruti components including the scheduler.

5 Conclusion

We feel that the technique of partial evaluation is appropriate for hard real-time programming. In contrast to many current hard real-time programming languages, it will allow the use of high-level programming language features to develop reusable programs that are not dedicated to a specific environment.

The distinction between compile-time and run-time variables in the language is useful. It helps (and forces) the programmer to write programs where execution time can be predicted at compile-time from the available environment information. These programs will have bounded execution times and fatal errors such as unbounded execution time will be caught at compile-time.

Acknowledgements

We would like to thank John Gannon and Satish Tripathi who provided valuable comments.

References

[Ber90] Andrew A. Berlin. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer Magazine*, 23(12):25–37, Dec. 1990.

[Ers82] A.P. Ershov. Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science*, 18:41–67, 1982.

[GMT85] Carlo Ghezzi, Dino Mandrioli, and Antonio Tecchio. Program Simplification via Symbolic Interpretation. In *Foundations of Software Technology and Theoretical Computer Science*, pages 116–128. LNCS 206, 1985.

[JSS85] N. D. Jones, Peter Sestoft, and Harald Søndergaard. An Experient in Partial Evaluation: The Generation of a Compiler Generator. In J.P. Jounnaud, editor, *Rewriting Techniques and Applications*, volume LNCS 202, pages 124–140. Springer Verlag, 1985.

[KS86] E. Kligerman and A.D. Stoyenko. Real-Time Euclid: a Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, pages 941–949, Sep. 1986.

[LA90] Shem-Tov Levi and Ashok Agrawala. *Real Time System Design.* McGraw Hill, 1990.

[Mea89] Al Mok and et. al. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *6th Workshop on Real-Time Operating Systems and Software*, pages 74–80. IEEE, May 1989.

[NP90] Vivek Nirkhe and William Pugh. Denotational semantics of the partial evaluator for mpl. Technical report, Department of Computer Science, University of Maryland, College Park, 1990. Under Preparation.

[NTA90] Vivek Nirkhe, Satish Tripathi, and Ashok Agrawala. Language Support for the Maruti Real-Time System. In *11th IEEE Real-Time Systems Symposium*, pages 257–266, Dec. 1990.

[PK89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Times of Real-Time Programs. *The Journal of Real-Time Systems*, 1(2):159–176, Sep. 1989.

[PS90] ChangYun Park and Alan C. Shaw. Experimenting With A Program Timing Tool Based On Source-Level Timing Schema. In *11th IEEE Real-Time Systems Symposium*, pages 72–81, Dec. 1990.

Predictable Real-Time Caching in the Spring System[1]

Douglas Niehaus, Erich Nahum, John A. Stankovic
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

1 Introduction

Designers of real-time systems, in common with all designers, seek the best possible performance. For conventional systems this generally means the best average case performance. In real-time systems, however, average case performance is not sufficient because the correctness of the system's results depend on *when* they are produced as well as *what* the results are. Real-time system designers must thus be able to *predict* the behavior of the system at development time in ways which are unnecessary for conventional systems. The predictions most often used are those for *worst case execution time* (WCET), since the designers are usually concerned with guaranteeing the correctness of the system's behavior under all possible circumstances. As a result, calculation of WCET for programs is an active research area [10][1][3][5].

Estimation of WCETs must consider the properties of the system within which the programs will be executed. The most obvious factor is the execution times of processor instructions, but other system properties are equally if not more important. These include the system's methods for handling interrupts, how it controls process's access to shared resources, the properties of the system's scheduling paradigm, and the presence of caches. A WCET estimate is *valid* if it is greater than or equal to the *actual* WCET. A WCET which is *less than* the actual WCET is obviously wrong. The ultimate goal, perhaps unattainable, is to produce WCET estimates which are *exact*. Failing this, we try to produce estimates which are valid but not too pessimistic.

This paper addresses the problem of how to make valid predictions about the effects of caching on the system so that the predicted WCET can be less pessimistic. The cache's effects, and our ability to predict them, depends strongly on the scheduling paradigm around which the system is designed. We will look at the rate monotonic and Spring paradigms, and discuss how the differences in their properties affect our ability to consider caches in WCET calculation. We will then discuss to what extent we can make predictions about the effects of caches on the system, and the extent to which these predictions can be used to reduce the pessimism of WCET estimates while preserving their correctness. Finally, we describe some of the open problems we are considering.

[1]This work is part of the Spring Project at the University of Massachusetts funded in part by the Office of Naval research under contract N00014-85-K-0398 and by the National Science Foundation under grants DCR-8500332 and CDA-8922572.

2 Scheduling Paradigms and Program Translation

Real-time programs execute on real-time systems, whose properties are the result of a plethora of design decisions. Among these, one of the most fundamental aspects of the system is the set of assumptions made by its scheduling paradigm. Programs are specified using the programming model, and their execution is managed using their run-time representation. Both the rate monotonic and Spring approaches to scheduling assume a process based programming model, but they assume different run-time representations.

The process model assumes that processes run until blocked for one of several reasons; resource contention, communication delays, scheduling preemption, or external interrupts. The rate monotonic research uses the process for both the programming and run-time models. Under rate monotonic scheduling processes run to completion unless blocked by resource contention or preempted by a higher priority process [6] [12]. Research in WCET estimation generally assumes the process run-time model and is thus conducted within its limitations, particularly the assumption of preemption at arbitrary times [5] [10] [1].

The Spring project assumes the process model for programming, with processes having separate address spaces [9], but the Spring scheduling paradigm assumes a very different run-time model. This significantly changes the properties of the system and thus the context within which WCETs are estimated. The Spring scheduling paradigm considers ***tasks*** with specified WCETs and resource requirements, and constructs an explicit plan for executing these tasks in a way that satisfies all deadline and resource constraints [14]. The Spring scheduling paradigm thus *avoids* blocking due to resource contention and can easily support the assumption that tasks execute without preemption, while the design of the Spring system isolates application code from external interrupts [13].

These properties are desirable, but require a significant effort to translate between the programming and run-time representations. The programming model describes computations in terms of processes contending for resources and blocking when necessary, while the run-time system represents them to the scheduler as groups of non-blocking tasks with known resource requirements. The translation process takes a reasonably novel approach with potentially significant properties, and is the subject of thesis research described elsewhere [8]. This paper considers how the translation can be used in the context of the Spring system to calculate WCET estimates in the presence of caches.

We will, however, describe the general outlines of the translation process. During translation, points at which the process may suspend are called *scheduling points.* Such points appear at the beginning and end of critical sections, at synchronous communication calls, or where explicit suspend calls appear in the code. When the process is running, each episode of its execution begins and ends at a scheduling point. The translation method is based on first producing a minimal size representation of the process's structure including these scheduling points, and then analyzing this representation to determine the WCET and resource use of each execution episode.

We call our representation a *time graph* since it combines the control flow structure of the process's basic block graph with the execution time for each block. The original time graph is isomorphic to the basic block graph used by the compiler for emitting code, and is reduced to minimum size using *subgraph reductions.* These reductions identify portions of the time graph which can be replaced with a single node containing the WCET of the original

subgraph. A simple example is the reduction of the subgraph for a conditional statement from three nodes, a node each for the test and the two branches, to a single node containing the sum of the test time and the maximum time of the two branches. These reductions can be modified to consider caches, to some extent, as they simplify the process's time graph. Subgraph reduction proceeds until further reduction is impossible.

Processes without internal scheduling points reduce to a single node, since they have only a single execution episode. Processes containing critical sections or other scheduling points will reduce to larger graphs, reflecting the fact that the process will exhibit more than one execution episode. The analysis of the minimal time graph determines the WCET and resource use of each execution episode. The worst case episodic behavior description is then given to the scheduler as the group of tasks representing the process. A number of interesting structural, scheduling, and computational issues arise, but they are discussed elsewhere [8]. In this discussion the important point is that the execution episodes will not be interrupted.

3 Caching in the Spring System

As a process executes under a traditional priority driven scheduling scheme it can be interrupted by external events, be preempted by a higher priority process, or block for resources at the start of a critical section. These events cause context switches, making it difficult to predict the hit rate of a cache in the system. This is precisely the run-time environment of systems using priority based scheduling, including those using the rate-monotonic approach. Research on predicting cache hit rates for these systems has eliminated some sources of uncertainty by allocating sections of the cache to tasks for the duration of their execution[4], which eliminates the cold start problem after context switches. However this research has not addressed all of the issues required to produce valid WCETs for real-time tasks in the presence of caches. The problem lies in determining the worst case path through the process code, which depends on the cache's properties.

The Spring paradigm for system design eliminates the sources of arbitrary interruption. Tasks are shielded from external interrupts in a Spring node by the use of dedicated application processors [13]. We assume a scheduler which constructs execution plans where tasks in the plan do not require preemption. Finally, tasks do not block for resources, because the schedule is constructed to avoid resource conflicts. As a result, we can consider conventional cache designs when making valid WCET predictions. As we consider ways to make our predictions less and less pessimistic, we may well wish to add features to our cache design. However, we believe a direct mapped virtual address instruction cache provides an interesting starting point. Data caching might be done, but is not considered here. We believe that its benefits will tend to be limited, though we will certainly consider it in our future work.

The benefits of our approach arise in several ways. First, the use of a virtual cache decreases the reference time for a hit, since cache processing can take place in parallel with address translation. Since context switches happen only at the large granularity of task boundaries, we can flush the cache at every context switch, and still gain a significant benefit from its presence. Flushing the cache at each context switch eliminates the aliasing problems commonly associated with virtual caches, simplifying the design. Direct mapped caches have several attractive properties including; generally lower cost, faster response, and an easily understood replacement policy. The rest of this section explains our simple methods

of including the cache in our WCET calculations, and the extent to which they can improve on the WCET estimates for a system without a cache, while maintaining their validity.

3.1 Worst Case Time Calculation

For this discussion, we will consider a simple architecture with a CPU, instruction cache and main memory connected by busses, but with the bus between the cache and main memory wider than that between the cache and the CPU. We will also assume a memory speed of $100ns$, a cache speed of $10ns$, an average of four instructions per cache line, and a cache miss penalty of from $100ns$ to $300ns$. These are reasonable values taken from discussion [2].

Our WCET calculation method considers instruction caches in two contexts; instruction prefetch and loops. When the CPU prefetches an instruction, it specifies an instruction address. If this instruction is not already in the cache, the block containing it will be retrieved. For sequential code execution, three of four instruction fetches will thus be cache hits. Assuming a cache miss penalty of $200ns$, we would thus have an execution time of $230ns$ with the cache, as opposed to $400ns$ without it; an improvement of 42.5%. Non-sequential code will, of course, reduce this. Our WCET calculation method can consider this effect quite easily. Recall that we create the original time graph from the basic block graph used for code emission. Each node in the time graph will note the number of instructions and bytes represented by the corresponding basic block, as well as its uncached execution time. During subgraph reduction, the linear portions of the graph are easily identified, and the speedup from cache prefetching accounted for.

The other way we can take caching into account is by noting when loops can be entirely contained in the cache. It is important to note that under our programming model, an upper bound on the number of times through the loop is necessary. This situation is slightly more complex, and can be divided into three cases. The simplest case is when the body of the loop is entirely *sequential*. The body of the loop will have been reduced to a single node giving the total number of bytes, instructions, and uncached execution time. The total number of bytes occupied by the loop is easily calculated at this point, and will either fit in the cache or not. For a direct mapped cache, the starting and ending addresses are not important. We only depend on the common property that the compiler will generate contiguous code for the loop. In this case the time for the loop is the uncached time for the first iteration, but the fully cached time for all subsequent passes.

The second case, when the body of the loop contains branches, is more difficult because the CPU will follow only one of the conditional branches on any given pass. It may, therefore, take several passes to bring the code for the entire body of the loop into the cache. This is illustrated by the cached and uncached times for the two branches of a conditional in the body of a loop. Call these branches A and B. Let us assume that the uncached and cached times of A are greater than the corresponding times of B, but that the *cached* time for A is *less than* the *uncached* time for B.

Now consider the WCET on each iteration of the loop. On the first iteration the uncached time for A is clearly part of the WCET. However, on the second pass the code for this branch is cached, and the uncached branch B is on the worst path. On all successive iterations, the cached time for A is on worst path. This clearly illustrates that the worst case path through

the code is *different* for the cached and uncached cases. Simply considering the cached time for the uncached worst case path would have produced an *invalid* WCET estimate. This also shows that the subgraph reductions must take such execution scenarios into account when calculating the WCET for the loop as a whole. This turns out to be fairly easy to do, and gives some idea of the flexibility of the subgraph reduction calculation method.

The third case is when the body of the loop contains subroutine calls. For a direct mapped cache, this could invalidate the caching of the loop body, since the footprint of the subroutine might overlap that of the loop body. If we are unable to consider how the subroutine code will act within the cache, we will have to assume the loop is *not* cached, which will increase the pessimism of our estimate. However, we can ensure no overlap in two ways. First, the code for the subroutine could be "inlined", which would convert the problem to the previous case. Inlining is appropriate for loops requiring optimization where the space penalty is acceptable, and is currently a feature of many compilers to permit saving subroutine call overhead. The second method would require assigning virtual addresses to the loop and subroutine code so that they would not overlap. This would require modifications to the loader to permit this kind of manipulation. Some investigators have taken this approach for conventional systems [11][7], but are considering the effects on average case performance. We will be developing approaches that address the effects on WCET.

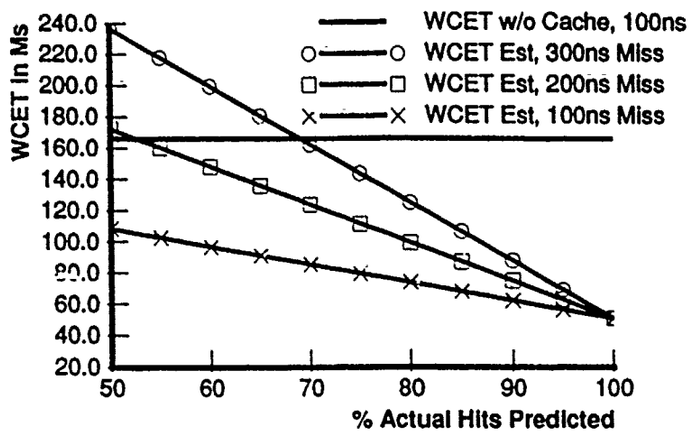
It is important to note that the subroutine footprint problem could also be affected by the associativity of the cache. At this time we are interested in seeing how much benefit can be derived from the simplest cache design, but a set associative or fully associative cache are alternatives that we can investigate if the simplest design proves insufficient. However, even with a fully associative cache, subtle control problems will arise, indicating that we may eventually wish to use a custom cache design in order to calculate less pessimistic WCET estimates.

3.2 Potential Benefits

One of the problems with evaluating real-time systems is the lack of representative programs. However, as an illustrative computation, let us consider multiplying two square matrices of size 50 on a machine with the properties discussed in the previous section. This is a triply nested looping calculation with a deterministic data path, and is a reasonable model for one class of computationally intensive real-time application activities. We wrote it in C, and produced the MIPS R3000 assembler code using the "-S" option of the compiler. The nested loops broke into seven basic blocks, of which five were non-trivial.

The example in Figure 1 illustrates several important points. It is important to note that since the data path is deterministic the hit rate *can* be determined by simulation of the cache on an address trace, and is thus a case for which the approach in [4] *is* sufficient to determine the worst case hit rate. If there were more than one path through the code, this would not be true. Further, since we have assumed that all the code fits in the cache, this is a situation where the cache provides a significant advantage. Other situations would not necessarily show such a marked improvement.

The horizontal line on the graph gives the execution time for the code on a machine without a cache, using just the $100ns$ memory access time. The three slanted lines give the execution times under various combinations of cache miss penalty and percentage of the




```
for(i=0; i<50; i++) {
  for(j=0; j<50; j++ ) {
    result = 0;
    for(k=0; k<50; k++) {
      result += matrix1[i][k]*matrix2[k][j];
    }
    matrix3[i][j] = result;
  }
}
```

Figure 1: Execution Time vs. Predicted Hit Rate and Test Code

actual hit rate predicted. These lines rise above the horizontal for low hit rates because of the cache miss penalty. This long recognized effect has important implications for real-time system design, since the WCET estimate must be based on the hit rate we can reliably *predict* for the system to support guarantees.

The justifications for including a cache in conventional and real-time systems thus differ in which hit rate is relevant. Conventional systems can use the actual rate, while real-time systems must use the predictable rate. The figure shows that unless we can predict a significant percentage of the *worst case* hit rate, we could actually *decrease* the number of processes the system could guarantee by adding a cache. This highlights the fact that principles which are familiar in conventional systems can have significantly different implications for real-time systems.

The graph shows that any method which tries to predict WCET time by analyzing the code *must* be able to identify significantly more than half of the actual hits in order to give a significant benefit. The actual percentage of the hit rate is impossible to determine without analyzing specific cache and process code properties, but we believe an adequate level of prediction is possible. For example, with the simple prefetch approach, we can easily predict a hit rate of 75% for the sequentially executed code. The frequency of sequential instructions for a RISC machine given in [2], is 85%. This would lead us to expect a hit rate of roughly 60% without taking loops into account. This prediction level is roughly that required to make the predicted WCET with the cache equal those without it for the $200ns$ miss penalty case. However, our ability to handle the looping case, particularly subroutine calls, is crucial to providing predictions of WCET that will permit us to realize a significant benefit from adding caches to real-time systems.

4 Summary and Future Work

This paper has discussed the problem of predicting WCET in the presence of caches. We have also given some idea of why the scheduling paradigm underlying the system design has a significant influence on how caching effects can be predicted. We gave the basics of our method for computing WCET, and some illustration of how it can be used to consider caching effects. The significant problem of predicting cache hits inside loops with subroutine

calls remains an open problem which will be addressed in our future work. As the work progresses, several interesting approaches are possible including: manipulation of the code layout in logical address space to improve the predictable speedup, use of a fully associative cache, and customization of the cache design. Beyond this, cache coherence will be an issue for data caching in multiprocessor systems, and we are interested in how very large caches would influence system design.

The ability to correctly estimate the WCET of a program is basic to all real-time scheduling paradigms, and is thus a question that must be answered adequately. The potential performance benefits of caching are so substantial that we must develop ways to use them predictably.

References

[1] P. Amerasinghe. An Interactive Timing Analysis Tool for the SARTOR Environment. Master's thesis, University of Texas at Austin, 1985.

[2] J. Hennessey and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[3] K. Kenney and K. Lin. Structuring Large Real-time Systems with Performance Polymorphism. In *IEEE Real-Time Systems Symposium.* IEEE, December 1990.

[4] D. Kirk and J. Strosnider. SMART(Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000. In *Proceedings of the IEEE Real-Time Systems Symposium.* IEEE, December 1990.

[5] E. Klingerman and A. D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, September 1986.

[6] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *JACM*, pages 46–61, February 1973.

[7] S. McFarling. Program Optimization for Instruction Caches. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[8] D. Niehaus. Program Representation and Execution in Real-Time Multiprocessor Systems. Phd. Thesis Proposal, University of Massachusetts-Amherst, 1991.

[9] D. Niehaus, C. Kuan, and J. Stankovic. Spring System Programming and Run-Time Models. Technical report, Spring Project Documentation, 1990.

[10] C. Park and A. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. In *IEEE Real-Time Systems Symposium.* , December 1990.

[11] K. Pettis and R. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, 1990.

[12] R. Rajkumar, L. Sha, and L. Lehockzy. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE, 1988.

[13] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *Special Issue of OS Review*, 23(3), July 1989.

[14] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transactions on Computers*, pages 949–960, August 1987.

STATIC ANALYSIS OF TIMING PROPERTIES FOR DISTRIBUTED REAL-TIME PROGRAMS*

Horst F. Wedde, Bogdan Korel, Dorota M. Huizinga
Computer Science Department
Wayne State University
Detroit, MI 48202

Abstract

In this paper a static analysis approach for verifying timing properties of real-time distributed programs is presented. We concentrate on the worst case response time of concurrent tasks which run independently but share logical or physical devices. For such tasks, prediction of the worst case response time involves estimation of time spent waiting for synchronization events. In particular, we investigate the class of Client-Server distributed programs in which independent, time-critical tasks (Clients) are synchronized only through additional Server tasks, playing the role of monitors or resource managers. This model follows the real-time design guidelines proposed to enhance schedulability and synchronization analysis. The analysis technique is flow graph oriented and leads to generating reduced program paths each of which represents a sequence of ordered local and global operations. While local operations are completely independent and can be performed concurrently, global operations require mutually exclusive access to shared logical or physical devices. For each operation an estimate of the minimum and maximum execution times is used for the actual worst case analysis in this model. We show that the problem of evaluating the worst case waiting (blocking) time is NP-complete. We show further that a previously held conjecture about NP-completeness of a reduced problem is wrong, by giving a polynomial algorithm for its solution of which we have proven the correctness. This solution provenly provides for a good upper bound of the original analysis problem. The effectiveness and complexity of the method are discussed.

I. Motivations

Recent developments in hardware and software technology have led to considerable progress in distributed real-time computing. This new and quickly evolving field lacks any well-established testing and verification methods, especially those which would explicitly take into account timing aspects of the system. The need for automated analysis tools derives from at least two different, although not unrelated concerns. First: the necessity for reducing testing costs, which account for up to 50% of the software development process [Mye79]. Second: increasing system complexity which renders manual testing all

*This work was partially supported by IBM Endicott (Research Agreement No. 6073-86) and by General Dynamics Land Systems (#DEY-605089).

but impossible. Automated or semiautomated tools would enhance chances of selecting appropriate test cases and result in producing more reliable and dependable software. These techniques are crucial for real-time computing especially, when a probabilistic approach seems to be unsatisfactory [Sta88]. The necessity of producing predictable timing behavior derives from the applications of real-time computing in which time errors might have very costly or even catastrophic consequences.

II. Related Work

Several formal models for specification and analysis of real-time programs have been proposed recently. These include RTL (Real-Time Logic [Jah86], [Jah87]), RTTL (Real-Time Temporal Logic [Nar88]), and some other extensions of temporal logic like Interval Logic ([Sch83]) or Metric Temporal Logic [Koy90]. The focus of these studies is on determining timing (and other) assertions of the system with respect to the specification expressed in the corresponding formal model formulas, rather than on the timing analysis of distributed programs.- We took a complementary approach to timing analysis of distributed programs in which verification of timing properties is separated from verification of logical correctness of the program. The idea was first suggested in [Wir77], and recently has been discussed in [Hsi89] and [Shc88].

The work reported in [Sha89] and [Pus89] utilizes the same concept of analysis as our own. Both studies describe high level language approaches to reasoning about time constraints (including the maximum execution time) of sequential programs. However, the analysis techniques presented are limited to singular programs, and possible blocking time due to shared data or resources is not considered. In [Sto87] the optimal worst case blocking time for resource contention is calculated using *frame superimposition*. The frame superimposition method shifts frames exhaustively, for every time unit, process, and combination of frames possible. Although very effective, this technique may well be infeasible for large systems due to its complexity. In [Lei82] a polynomial time algorithm for finding an approximate upper bound on the worst case blocking time has been proposed. Our method outperforms the one reported in [Lei82] with respect to both the actual results (i.e. it finds a better upper bound on the blocking time) and the algorithm complexity.

III. System Model.

Following the Ada real-time design guidelines [Sha90] we limit our studies to the Client-Server type of distributed programs. Our system consists of a set T of time-critical tasks (e.g. Ada-like tasks) which we call Client tasks and a set S of Server tasks (e.g. Ada rendezvous) which play the role of monitors or resource managers. Each task runs on a designated logical or physical processor; however for analysis purposes we assume a pure maximal physical parallelism. Server tasks run in infinite loops and accept Clients according to a FCFS rule. (This assumption can be easily generalized for any priority driven policy, like priority inheritance or priority ceiling.) All other loops are assumed to have a known and bounded maximum number of iterations. The programs follow structural coding guidelines which allow for easy reductions of corresponding flow graphs. In addition for each time critical task, there is a time frame (minimum and maximum time) representing the time interval within which the task starts.

IV. Analysis Technique.

This paper is an extension of our previous work reported in [Wed91] in which the system model consisted of a singular Server task and several Client tasks. Subsequently we sketch

a flow-graph reduction technique which leads to the generation of reduced paths for each task. Details of this technique as well as a correctness proof of the reduction procedure can be found in [Wed91]. More details will be offered in the final paper.

Flow Graph Reduction.
A distributed Ada-like task system defines a control flow graph (or forest) in which each basic ***block*** [Hec77] (a group of sequentially executed ***simple statements***) is represented by a corresponding node and edges represent possible transfer of control between blocks. For each task t there is a designated initial node N. Our time oriented analysis first refines the control flow graph and then reduces it to create a ***reduced graph***. The refinement procedure consists of "splitting" each node of the flow graph into two nodes representing the beginning and the end of the corresponding block and adding a new edge between these two. This edge is labelled with a pair representing the best and the worst case execution time of the basic block. The best and the worst case execution time for any basic block is estimated by repeated module execution. The reduction algorithm of the refined graph eliminates all internal loops (i.e. the loops which do not include any synchronization statements) and it works by replacing innermost cycles with singular edges and expanding outward. The singular edge weight is set equal to the pair representing the best/worst case execution time of the loop. Loops which include synchronization statements are eliminated by the repetition of the (already reduced) body of the loop. For each remote server call (remote rendezvous) a bounded communication delay is added.

The above procedures having been applied successively to a distributed Client-Server type of program create a corresponding acyclic directed graph (or forest) with a set of designated nodes N representing the beginning of each task, and a set of designated nodes M representing the end of each task. Edges in the reduced **Directed Acyclic Graph *(DAG)*** correspond to the control flow in the original program and they are labelled with the minimum and maximum execution time of the basic blocks.

Path Oriented Timing Analysis.
A set of reduced paths, one for each involved task, is generated form the DAG. Each of the reduced paths consists of an ordered sequence of local and global operations and it is called a task-oriented reduced path. Each operation is labelled with a pir corresponding to the best and the worst execution time of it. In particular, we choose the task for which the worst case analysis is to be performed. Such a task starts within its time frame and performs its local and global operations according to the order described by its path. The worst case response time of the task depends on local operations as well as the waiting (blocking) time due to global operations. If two or more tasks submit their requests for the same global operation at the same time, we assume that the task of interest will be served last. This assumption guarantees the worst case response estimation. Consideration of starting time frames, Server request orders and times of local and global events make timing analysis very accurate but increases problem complexity. Subsequently we define blocking rules and the general worst case blocking (waiting) time problem.

FCFS Worst Case Blocking Time Rules.
Task T_i requesting access to server S is blocked at S by task T_j iff:

1. T_j requested access to S *before* T_i and it is either still waiting for the service or it has been granted access to S but it did not complete yet. (FCFS requirement.)
2. T_j requested access to S "simultaneously" with T_i. (T_i will be served after T_j - worst case requirement.)

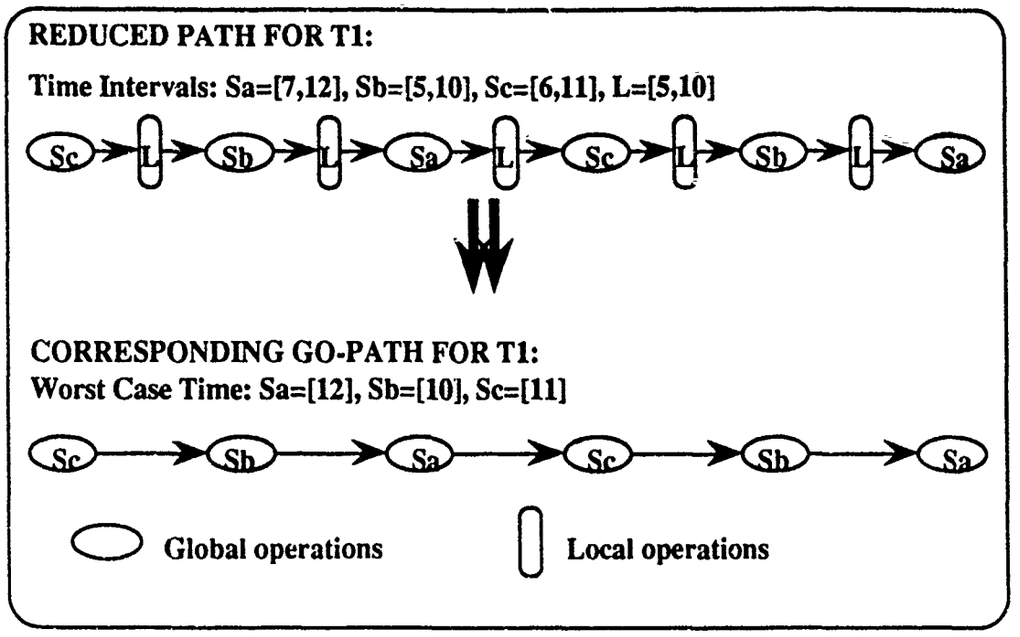


Figure 1: Example of a Reduced Path and the corresponding GO-Path

Worst Case Blocking Time (WCBT) - General Problem.
Given a set of n tasks $T=\{T_1,T_2,...,T_n\}$represented as a set of task oriented reduced paths $P=\{P_1, P_2,...,P_n\}$ (each P_i consisting of a sequence of local and global operations with an execution time interval assigned to each operation), starting time frames for each task: F_1, $F_2,...F_n$, and designated task T_i from T, find the worst case blocking time of T_i by tasks $T_1, T_2,...T_{i-1},T_{i+1},...,T_n$, assuming that paths $P_1, P_2,...,P_n$ are executed.
Note that the solution to the WCBT-general problem consists of:

1. The starting times $st_1,st_2,...,st_n$ for each task, s.t. st_i is in F_i.
2. The acceptance order for "simultaneous" requests (for the tasks other than T_i)

Theorem 1: The WCBT-general problem is NP-hard. Moreover the problem remains NP-hard even if time interval are reduced to single points (operations take constant amount of time) and are all equal.

We will give an outline of the proof in the paper.

Our theoretical findings about evaluating the worst case blocking time led us to the formulation of the "relaxed" version of the WCBT problem.The relaxed problem disregards the starting time frames and the times between server requests, however it preserves the order of requests and the execution times of servers. Subsequently, we define the relaxed version problem and show the polynomial time dynamic programming algorithm which solves it.

First we define a task oriented global operation subpath of P_i (called ***GO-path P_i)*** to be a subsequence of all global operations of P_i with the worst case execution time of each operation (see Fig.1).

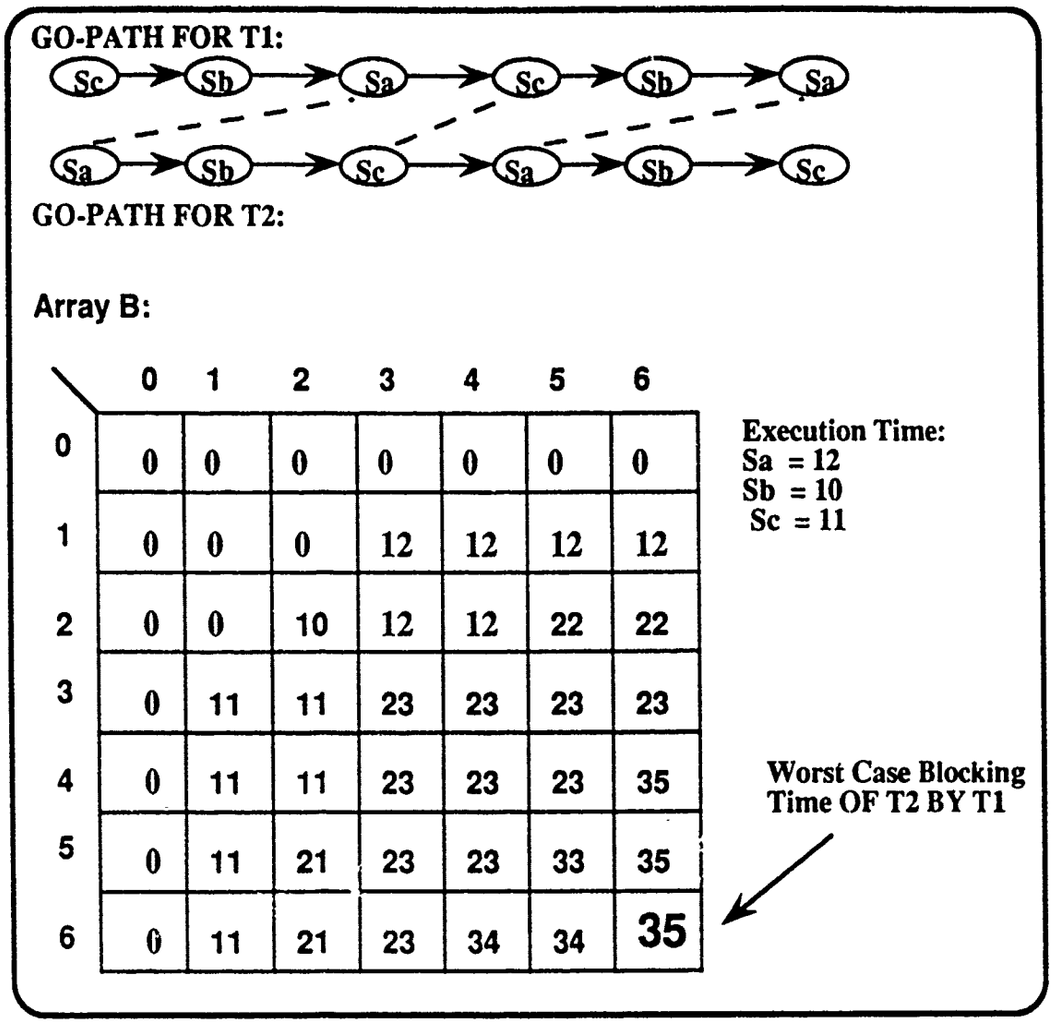


Figure 2 Worst Case Blocking Time of T2 By T1.

Worst Case Blocking Time - Relaxed Version Problem.
Given a set of n tasks $T=\{T_1,T_2,...,T_n\}$ projected into a set of GO-paths $P=\{P_1,P_2,...,P_n\}$ (each P_i consisting of a sequence of global operations with the worst case execution time assigned to each operation), and designated task T_i from T, find the worst case blocking time of T_i by tasks $T_1, T_2,...,T_{i-1},T_{i+1},...,T_n$ assuming that the GO-paths $P_1, P_2,...,P_n$ are executed.

It was a conjecture that the WCBT-relaxed version problem was NP-hard for n=2 [Lei82]. In contrast to this, we have

Theorem 2: There is an algorithm which solves the problem in polynomial time for arbitrary n.
A proof will be outlined in the paper.

Furthermore: The solution of the WCBT-relaxed version problem, can be utilized as a good upper bound for the general problem. The idea of the solution originated from the Longest Common Subsequence problem [Hir75].
To simplify the description of the algorithm we assume that T_1 is the task for which the worst case blocking time is to be found and that all GO-paths are of the same length k. (None of this assumptions is needed for the algorithm to work.)
Let each GO-path P_i be represented an ordered sequence of records each consisting of two fields: one containing the name of the server and the second containing the worst case execution time of the server. Therefore, for 1<=l<=k, we have:

P_i[l].server_name - name of the l-th global operation in G0-path P_i
P_i[l].server_time - worst case execution time of the l-th operation in GO-path P_i

```
Function Block(X,Y: record)
(* returns server_time if X and Y contain the same server name, otherwise it returns 0 *)
begin
        if ( X.server_name=Y.server_name)  then
                return(X.server_time)
        else
                return(0);
end;

Function  Initialize(B)
(*initializes blocking array called B*)
begin
        for l:=0 to k do
                B[l,0]=0;
        for l:=0 to k do
                B[0,l]=0;

end;
```

Function MaxBlock(P_1,P_2,...P_n: GO-paths);
(* returns the solution WCBT-relaxed version problem for task T_1 *)

```
begin
1   MAXB:= 0;                   (* maximum blocking time *)

2   For r:=2 to n   do          (* for all paths other than P1 *)
3       Initialize (B);
4       For l:=1 to k do
5           For m:=1 to k do
6               B[l,m]:=max(B[l-1,m], B[l,m-1], B[l-1,m-1]+ Block(P1[l],Pr[m]));
7           end for;
8       end for;
9        MAXB:=MAXB+B[k,k];
10  end for;

11  return( MAXB);
end;
```

Function MaxBlock calculates the worst case blocking time of T_1 by tasks $T_2, T_3, \ldots, T_n$ (lines 3 to 8) and sums the results (line 9). We have shown that the above procedure correctly calculates the solution for the WCBT-relaxed version problem.The order of complexity of MaxBlock is $n*k^2$ in the size of the problem encoding. It outperforms in both accuracy and time complexity the results published in [Lei82] in which a heuristic algorithm for the WCBT-relaxed version has been developed.
The results of MaxBlock are then utilized to calculate an upper bound on the actual task response time.

V. Conclusions and Future Work.

In this paper an approach to automated analysis of distributed real-time programs has been presented. We have concentrated on the timing criterion of system correctness for which no automated verification methods or tools exist, to our knowledge. The method described is based on the static analysis of the task system and generation of GO-paths (global operation paths) for which actual timing analysis is applied. We have shown that a good upper bound on the worst case blocking time can be found using low complexity procedures, however the optimal solution is intractable.- In our further research we intend to generalize the WCBT -reduced version problem flow graphs (or at least flow trees) to reduce the lengthy process of repeated path generation.

References.

[Hec77] M.S. Hecht; *Flow Analysis of Computer programs*, Elsevier North-Holland, 1977.

[Hir75] D.S. Hirschberg; A Linear Space Algorithm for Computing Maximal Common Subsequences; CACM, vol 18, No 6, pp. 341-349.

[Hsi89] C.S Hsieh, Timing Analysis of Cyclic Concurrent Programs, pp. 312-318.

[Jah87] F. Jahanian, A.K. Mok, A Graph Theoretic Approach for Timing Analysis and its Implementation, *IEEE Transactions on Computers*, vol 36, No 8 , 1987, pp. 961-975.

[Jah86] F. Jahanian and A.K. Mok, Safety Analysis of Timing Properties in Real-Time Systems, *IEEE Transactions on Software Engineering*, vol.12, No 3, 1986, pp. 890-904.

[Koy90] R.Koymans, Specifying Real-Time Properties with Metric Temporal Logic, Real-Time Systems, vol 2, No.4, 1990, pp.255-299.

[Lei82] D.W. Leinbaugh, M-R. Yamini; Guaranteed Response Time in a Distributed hard-Real-Time Environment; Proc. of Real-Time Systems Symposium; Dec. 1982, pp. 157-169.

[Mye79] G.J.Myers; *The arts of software testing*, John Wiley & Sons, 1979.

[Nar88] K.T. Narayana, A.A. Aaby; Specification of Real-Time Systems in Real-Time Temporal Interval Logic; Proc.Real-Time Systems Symposium, Dec.1988, Huntsville, Al., pp.86-95.

[Pus89] P.Puschner, Ch. Koza; Calculating the Maximum Execution Time of Real-Time Programs, *Real-Time Systems*, vol.1, No 2, Sept. 1989, pp. 159-176.

[Sch83] R.L.Schwartz, P.M. Melliar-Smith, and F.H. Vogt, An Interval Logic for High -Level Temporal Reasoning, Proc. 2-nd Annual ACM Symposium on Principles of Distributed Computing, 1983, pp. 173-185.

[Sch88] R.L.Schneider, Critical Issues in Real-Time Systems, Technical Report 88-914, Dept. of Computer Science, Cornell University (May 1988).

[Sha89] A.C. Shaw; Reasoning About Time in Higher-Level Language Software, *IEEE Transactions on Software Engineering*, vol.15, No 7, July 1989.

[Sha90] L.Sha, J.B. Goodenough; Real-Time Scheduling Theory and Ada; IEEE Computer; April 1990; pp.53-62.

[Sta88] J.A. Stankovic; Misconceptions About Real-Time Computing, *IEEE Computer*, Oct. 1988, pp.10-19.

[Sto87] A.D. Stoyenko, A Schedulability Analyzer for Real-Time Euclid; Proc. Real-Time Systems Symposium, Dec 1987, San Jose, Ca, pp. 218-227.

[Wed91] H.F. Wedde, B.Korel, D.M. Huizinga; A Critical Path Approach for Testing Distributed Real-Time programs; Proc. 24-th International Conference on System Sciences; Jan. 1991, Ha, vol 2, pp. 400-407.

[Wir77] N.Wirth; Toward a Discipline of Real-Time Programming, *CACM*, vol 20, No 8, 1977, pp.577-583.

An Integrated Approach to Monitoring and Scheduling in Real-Time Systems

Farnam Jahanian
Ragunathan Rajkumar
IBM Thomas J. Watson Research Center
Yorktown Heights

Abstract

In real-time systems, interactions with the physical environment can lead to unexpected conditions such as system overload and the missing of deadlines. It is highly desirable that these error conditions do not lead to total system failure and that the critical functions of the system are still performed. In addition, when an error condition arises, the system designer may prefer to invoke an error-specific operation to correct or recover from the error. The ability to detect violation of design assumptions or occurrence of unexpected conditions requires an integrated approach in which system timing constraints can be expressed and monitored, and appropriate action taken when a constraint is violated. However, the run-time monitoring of the system constraints consumes time and must not intrude upon the ability of the system to meet its constraints. Our approach allows the run-time monitor to be scheduled as a time-constrained activity and therefore can be part of the schedulability test for the system. This approach can then be further extended beyond simple timing constraints to complex timing assertions as well as the specification and monitoring of arbitrarily complex safety requirements.

1. Introduction

In designing real-time systems, we often make assumptions about the behavior of the system and its environment. For example, each task is assumed to have a worst-case execution time, and some asynchronous signals are assumed to have a minimum interarrival time. However, the unpredictable nature of the environment may not always satisfy these requirements, and any violations of these assumptions must not cause the system to fail. A limited amount of work has been carried out in dealing with potential violations of the design assumptions. For example, the ability of a system to perform critical activities under a transient overload is referred to as *stability* [8], and is a major motivation for use of the Rate-Monotonic Scheduling (RMS) framework. The concept of stability can be extended to possible hardware, software and operational failures.

Our approach is to express as invariants the design assumptions and system properties that must be maintained, and to monitor these invariants at run-time. If the violation of an invariant is detected, the system will perform an appropriate action to correct or recover from the error. Unfortunately, run-time monitoring of system constraints consumes resources and can intrude upon the normal timing behavior of the system. Our solution is to integrate the run-time monitoring with the scheduling methodology, by treating the monitoring activity as another real-time task to be scheduled. The resulting real-time system, therefore, would be predictable when there are no violations of design assumptions, and robust in their presence.

In addressing the implications of run-time monitoring, it is helpful to distinguish between the *language* and the *scheduling* issues. The language related issues concern the formal specification of properties and the mechanisms for testing the satisfiability of these constraints at run-time. The run-time constraints can represent, for example,

- Timing constraints ranging from simple deadline requirements to complex end-to-end timings,
- Fault tolerance requirements which may specify that in case of certain hardware failures or software conditions, a mode change (reconfiguration or a degraded mode of operation) must be initiated, or
- Execution order constraints such as precedence constraints on a set of events or actions.

The monitoring facility provides a real-time system designer with a host of capabilities. The programmer should

be provided a notation for specifying arbitrarily complex constraints which must be checked at run-time. This notation can be part of a programming language or it can take the form of a specification language (e.g., based on relation query language or logic) superimposed on top of an existing programming language. An important issue is how to establish a bound on the computational requirements in evaluating a condition at run-time. For example, calculating the overhead in determining whether a deadline has been missed is straight-forward. However, a powerful specification language may allow expression of complex constraints. Testing for a violation of a constraint at run-time may require remembering and examining a history of past events. Making a system predictable necessitates establishing a bound on the event histories that must be examined at run-time. The monitoring facility should also provide the capability so that a programmer has the option to specify conditions that can be tested either synchronously or asynchronously. Furthermore, the programmer should be able to specify special handlers to be executed when a certain condition is violated. If no handlers are specified, the system can perform a default action as well. Many other language-related issues arise which are beyond the scope of this paper.

A related set of issues concerns scheduling real-time tasks when a run-time monitoring facility is supported. Since monitoring constraints consumes resources, it is crucial to determine its intrusiveness on the normal activities of the system. For example, the monitoring activity itself can be treated as a set of tasks to be scheduled. Given a set of real-time tasks and a set of conditions to be monitored, the requirements imposed on the monitor can be determined and its worst-case performance can be guaranteed. More importantly, suppose that a methodology such as the rate-monotonic scheduling framework is used. Then, both monitoring and the application tasks can use the same scheduling mechanisms such that the benefits of predictability and analyzability of the framework will be available for both. In this paper, we shall address the issue of scheduling real-time tasks along with activities which monitor system events to detect and handle any violation of system requirements.

The rest of this paper is organized as follows: Section 2 reviews our event-based model for expressing and monitoring run-time constraints. Section 3 presents how the monitor described in Section 2 can be scheduled using the Rate-Monotonic Framework. Finally Section 4 presents the concluding remarks.

2. The Run-time Monitor Model

The run-time monitor model that we consider in this paper is based on the model proposed in [3, 2]. A system computation is a sequence of event occurrences. Informally, events represent things that happen in a system. An event occurrence defines a point in time at which a particular instance of an event happens in a computation. Timing assertions about a system can be expressed as relationships among these event occurrences.

Events can fall into one of two categories. *Label events* are used to denote the initiation and completion of a sequence of program statements. *Transition events* capture assignments of values to a particular type of variables. It may not be sufficient to remember only the last occurrence of each event to detect the violation of a timing property. An *event history* stores the times (and values for transition events) of a finite number of previous occurrences.

The model distinguishes between two general ways in which event histories can be utilized in specifying and monitoring timing assertions: *synchronous* vs. *asynchronous*. In synchronous monitoring, the programmer can explicitly check for the satisfiability of a property at a particular point in the execution of the program. This is done by directly manipulating the event histories which are shared by the cooperating tasks. Thus, testing and handling of any violation is carried out synchronously. This category of timing constraint is referred to as *embedded* constraints. As described in [3], an RTL-like notation can be used to specify embedded constraints in a program. The following code segment illustrates an embedded constraint.

```
/* temperature is a variable */
if (@val( temperature, i )  > threshold) {
    shutdown_nuclear_reactor();
}
```

The event histories are accessed by two RTL-like functions: the occurrence function *@(e,i)* which returns the time of the i^{th} occurrence of event *e*, and *@val(v,j)* which returns the value of variable *v* at its j^{th} occurrence.

Alternatively, in asynchronous monitoring, the constraint is enforced during the entire execution of the program. This category of constraints is referred to as *monitored* constraints. Thus, testing and handling of exceptions are performed asynchronously. The events generated by the tasks are sent to the system monitor (a separate task) which is responsible for maintaining the event histories. Whenever an event occurs that may violate the satisfiability of the constraint, the system monitor re-evaluates the expression and invokes the appropriate handler if the expression is violated.

A monitored constraint, where an acknowledgement must occur no later than 5 time-units after a send, can be expressed as shown in the following code segment. The notation *@(send, -1)* denotes the time of the most recent occurrence of the event *send.*

```
@(send, -1)≤@(ack, -1) ∧ @(ack, -1)≤@(send,-1) + 5
```

In synchronous monitoring, since the event history is shared, a synchronous access to the event history results in the caller being blocked. Thus, blocking can be caused both when writing to the event history or when testing a constraint. Asynchronous monitoring represents non-blocking testing where the testing done by the monitor is transparent to the programmer (except for the time penalty, which shall be addressed in Section 3).

This monitoring model proves useful in multiple ways. For example,

- Typically, a monitored expression would be used for detecting the violation of a timing constraint. For instance, it can be used to check the satisfiability of a deadline where the constraint can be simple as in the first code segment example, or could be complex such as an end-to-end deadline between events happening across multiple processors.
- These expressions can also be used to detect that a timing constraint will not be met at a later time. For instance, one can express the property that an intermediate point must be reached at a time that would make it possible to meet a later deadline. As a result, the system can detect that a deadline will be missed even before it happens and corrective action can be taken.
- It is also possible to express and enforce precedence constraints using this model. For instance, suppose that the i^{th} occurrence of event *B* must succeed the i^{th} occurrence of event *A*. Event *B* could represent in this case the beginning of a subtask with a precedence constraint while *A* could represent the end of another subtask. If the subtask generating *B* tests for the occurrence of event A, then the subtask will be blocked until the event *A* occurs, thereby achieving the desired effect.
- The asynchronous monitoring model can also be used to implement graceful degradation, reconfiguration or mode changes. Based on the occurrences of certain events, a system monitor can change the execution mode of a system to deal with unexpected conditions. Similarly, the synchronous monitoring model can be used to test an assertion in a program and change the execution of the task based on some event occurrences, for example, to evaluate an acceptance test (in a recovery block) before the task execution can continue.

Currently, a uniprocessor implementation of the model is operational at IBM Research [1]. A set of library routines in C provides the appropriate functions for specifying and monitoring assertions. The implementation, on AIX running on IBM RS/6000 workstations, hides the operating system dependencies in a well-defined layer on top of which the monitoring library is built. Error handlers are user-programmable and can, for example, trigger mode changes or a degraded mode of operation. A default handler is also provided by the system.

3. Scheduling Monitoring Activities

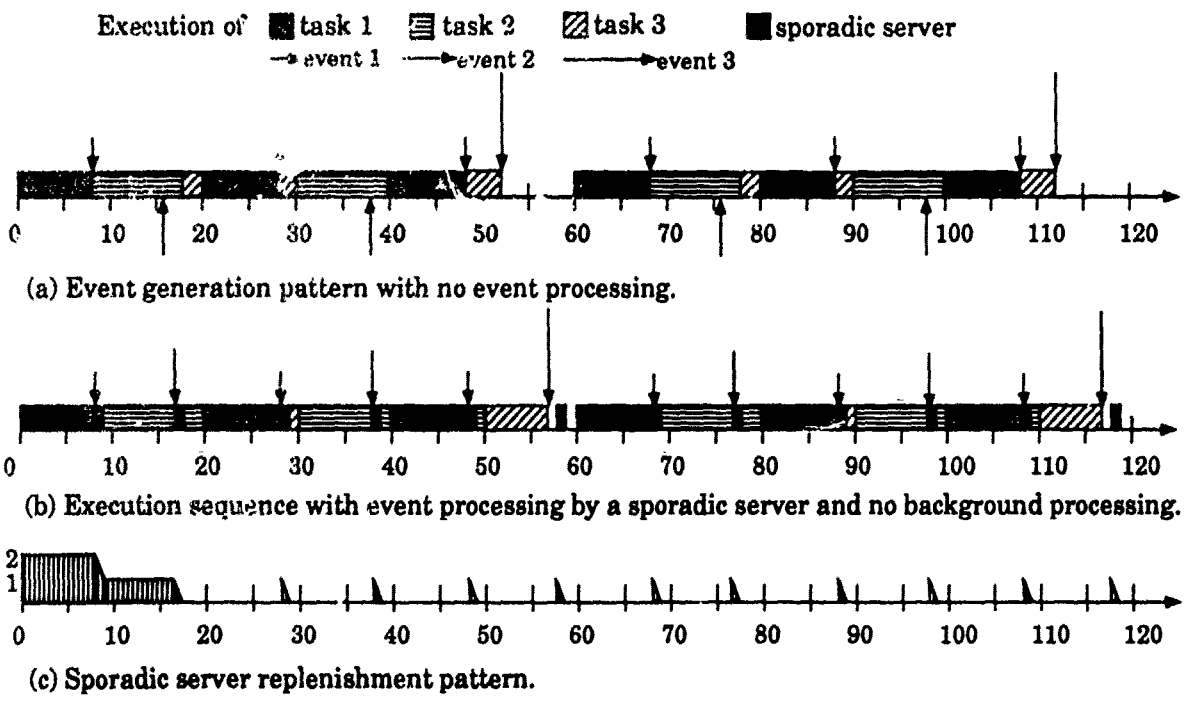
The monitoring model helps to detect violation of timing constraints synchronously or asynchronously. Its primary disadvantage, however, is that the maintenance of the event histories and the checking of events consume time, and therefore can be intrusive in a real-time system. Our approach is to treat the monitoring activity as a schedulable entity so that it can be addressed within the same framework as the real-time tasks running in the system. In scheduling theory, a common assumption is that each task has a known worst-case execution time. Similarly, it makes sense to assume an upper bound on the number of events that each task can generate and the time taken to test an expression or to update the event history. A related issue is the size of the history that must be maintained for each event. It has been shown in [3] that for a set of expressive subclasses of properties, an upper bound can be established for the history size for each event.

We now discuss scheduling issues on uniprocessor and multiprocessor systems, based on the Rate-Monotonic Scheduling (RMS) framework.

The monitor model can be implemented on both uniprocessors and multiple processor systems. However, the scheduling impact of an implementation depends heavily on the underlying platform and whether the monitoring is synchronous or asynchronous. The basic difference between synchronous and asynchronous monitoring from a scheduling perspective is that the monitoring overhead is imposed on the application task in the former and on the system in the latter. In either case, it is essential that the time consumed by monitoring be bounded and accounted for in the schedulability analysis of the system.

3.1. Uniprocessor Scheduling Issues

Asynchronous Monitoring: The scheduling issue to be addressed is that the instants at which the various events will be generated (even by periodic tasks) will not be strictly regular. The following example illustrates several features of the problem and our solution.



(a) Event generation pattern with no event processing.

(b) Execution sequence with event processing by a sporadic server and no background processing.

(c) Sporadic server replenishment pattern.

Figure 1: An Asynchronous Monitoring Sequence

Example 1: Suppose that asynchronous monitoring is used, and that there are three tasks with periods 20, 30 and 60 units, and execution times 8, 10 and 8 units respectively. Each task instance generates an event after 8 units of execution. The event generation sequence is illustrated in Figure 1.a. As can be seen, the events generated by task 1 and task 3 are periodic, with events of task 1 occurring at time 8, 28, 48, $\cdots$ and events of task 3 occurring at time 52, 112, $\cdots$. On the other hand, the events generated by task 2 (at time 16, 38, 76, 98, $\cdots$) do *not* correspond to exact periodic intervals. The event sequences of task 1 and 3 would also not be periodic if the stochastic execution nature of a task is taken into account. A server task which processes these events would therefore violate the behavior of a pure periodic task [5] and can cause deadlines to be missed earlier [4].

The solution to this "jitter" problem is the use of a sporadic server task [11] to process the asynchronous events. The sporadic server executes at a high priority, and is assigned a period as well as some computational time during that period. Any continuous computational time consumed by servicing monitor requirements, can be replenished a server's period after the beginning of this consumption.

The execution pattern repeats every hyperperiod which is 60 time-units (LCM of 20, 30 and 60) for this task set. Suppose that each event can be processed in 1 time-unit. A total of 6 events are generated during the hyperperiod, and hence a server task must have a capacity of at least 6 units every 60 time-units to process these events. The server task can be assigned the highest priority by giving it a period of 20 and a per period capacity of 2 units. The total utilization of the task set with the server is $\frac{2}{20}+\frac{8}{20}+\frac{10}{30}+\frac{8}{60}=0.96$, and a critical zone analysis shows that the task set is schedulable.

The execution sequence with the server, and the server replenishment profile are also shown in Figure 1.b and 1.c respectively. Note that at times 57, 117, $\cdots$, an event of task 3 has occurred, but the server does not have any capacity. Hence, the processing of the event takes place after 1 time-unit when the server's capacity is replenished by 1 unit. Events are processed immediately at the highest priority at all other times. If the sporadic server executes at background priority in the absence of capacity, this event could have been processed immediately as well. In other cases, additional capacity may have to be assigned to the sporadic server if an expression must be tested immediately after an associated event occurs.

If some expressions to be tested by the asynchronous monitor are mutually independent, multiple sporadic server tasks each responsible for one set of inter-dependent expressions can be created. If the possibility of an overload is present, events, expressions and handling of violations can themselves be prioritized. As a result, if an overload does arise, the more critical expressions would still be tested and the appropriate error handlers would be invoked. Given a sporadic server implementation, the model is relatively straightforward to implement on a uniprocessor.

Synchronous monitoring: This case is much simpler. The event processing time can be treated as part of the execution time of the task generating the event. The shared event history can be a source of blocking and a real-time synchronization protocol such as the priority ceiling protocol [9] can be used to access it. As a result, a task can be blocked for at most one event processing of a lower priority task, and the corresponding schedulability test can be carried out [9]. If blocking occurs on events that are yet to occur, its schedulability impact would be application-dependent and is beyond the scope of this paper.

3.2. Multiprocessor and Distributed Implementations

This section outlines possible extensions of the uniprocessor scheme to multiple processor systems. Additional requirements consist of

- a schedulable communication medium (such as a priority-based backplane bus or a token-ring) and

- a message-passing model to pass event histories between processors for asynchronous monitoring.
- a shared memory model or a remote procedure call model for synchronous monitoring.

In a multiple processor system, centralized, distributed or hybrid implementations are possible. In a centralized implementation, there is a a single monitor just like in the uniprocessor case, and events happening on other processors must be communicated to the central monitor. Clearly, this scheme involves more overhead and is inefficient. In addition, if a common clock is absent, clock variations (even for synchronized clocks on different processors) must also be taken into account and specified as part of the timing expressions. In a decentralized environment, a monitor would exist on each processor. During initialization, each monitor would query other monitors in the system and maintain a list of events being monitored by each. Whenever an expression has to be evaluated, the monitor attempts to perform the evaluation locally, if possible. If not, it contacts the monitors that maintain the required event histories to obtain the information not locally available. Alternatively, each monitor also obtains during initialization the expressions evaluated by other monitors. When a local event that is of interest to other monitors occurs, each monitor multicasts the event to the monitors which need access to this event history. As a result, when an expression has to be evaluated, a monitor would have all information locally. In a hybrid scheme, a local monitor on each processor *and* a global monitor would be present. When an event of external interest occurs, each local monitor performs a write-through to the global monitor. Therefore, when the evaluation of an expression requires external event history, only the global monitor needs to be contacted. A demand-based version of this scheme is also possible, where the global monitor acts as a repository of information about which monitor maintains which event history.

For asynchronous monitoring, the messages to be transmitted between distributed monitors must be scheduled such that they will be guaranteed to reach their destinations by specific points in time. As a result, it is possible determine when the event histories required to evaluate an expression would be available at a monitor. Naturally, the workload to be scheduled on the communication medium depends on the type of scheme adopted. Since the transmitted messages will not be exactly periodic, they would need to be scheduled again using a sporadic server. The reader is referred to [10] for a detailed schedulability analysis of such a message-passing paradigm. Finally, since events can be generated in physically separate processors, expressions *cannot* be tested as soon as they happen, and there will a time-lag between event occurrence and expression evaluation.

For synchronous monitoring, the multiprocessor priority ceiling protocol [7] may be used in shared memory systems, and the multiple processor priority ceiling protocol [6] may be used in distributed systems. The communication medium must still be scheduled using a sporadic server task.

4. Concluding Remarks

This paper described our current studies on integrating the notion of monitoring system behavior with scheduling real-time tasks. Our objective is to design real-time systems which are not only predictable in a benign environment but also robust in the presence of failures. We present how the monitor can be scheduled on a uniprocessor, and outline implementation schemes for multiprocessors and distributed systems. The approach illustrates that synchronous or asynchronous monitoring of constraints can be integrated with scheduling real-time activities of the system. The chief advantage of the approach is that the explicit scheduling of monitoring activities allows arbitrarily complex expressions to be evaluated at runtime.

References

1. Chodrow, S., Jahanian, F., and Donner, M. A Run-Time Monitor for Real-Time Systems. IBM Research Report, April, 1991.

2. Donner, M. and Jahanian, F. "RTL meets ORE". *Proc.of IEEE Workshop on Real-Time Operating Systems and Software* (May 1990), 55-61.

3. Jahanian, F. and Goyal, A. "A Formalism for Monitoring Real-Time Constraints at Run-Time". *Proc. of 20th Fault-Tolerant Computing Symposium* (June 1990), 148-155.

4. Lehoczky, J. P., Sha, L. and Strosnider, J. "Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment". *IEEE Real-Time System Symposium* (1987).

5. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

6. Rajkumar, R., Sha, L., and Lehoczky J.P. "Real-Time Synchronization Protocols for Multiprocessors". *Proceedings of the IEEE Real-Time Systems Symposium* (1988), 259-269.

7. Rajkumar, R. "Real-Time Synchronization Protocols for Shared Memory Multiprocessors". *The Tenth International Conference on Distributed Computing Systems* (1990).

8. Sha, L., Lehoczky, J. P. and Rajkumar, R. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling". *IEEE Real-Time Systems Symposium* (1986).

9. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers* (September 1990), 1175-1185.

10. Sha, L., Rajkumar, R., Locke, C. D. "Real-Time Applications Using Multiprocessors: Scheduling Algorithm and System Support". *Submitted for publication* (1991).

11. Sprunt, H. M. B. *Aperiodic Task Scheduling for Real-Time Systems.* Ph.D. Th., Carnegie Mellon University, August 1990.

New Paradigms for Real-Time Database Systems

Robert P. Cook, Sang H. Son, Henry Y. Oh, Juhnyoung Lee

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

1. Introduction

Real-time database systems (RTDBS) are database systems where transactions have timing constraints such as *deadlines*. The correctness of the system depends not only on the logical results but also on the time within which the results are produced. In RTDBS, transactions must be scheduled in such a way that they can be executed before their corresponding deadlines expire. For example, both the update and query in the tracking data for mission must be processed within given deadlines.

Conventional database systems are typically not used in real-time applications due to poor performance and lack of predictability. In other words, paradigms used in conventional database systems are not suitable in real-time database systems [Son90]. To address this problem, we have been investigating new database technology and paradigms for real-time systems using both theoretical as well as experimental approaches. They can be grouped into the following research tasks: (1) investigating new protocols for transaction scheduling, concurrency control, and checkpointing, and (2) developing experimental database systems that can provide real-time features over conventional relational databases. New scheduling and concurrency control protocols developed in the first task are being implemented in the experimental database systems and the prototyping environment for performance evaluation.

Our research effort in the area of real-time transaction scheduling has resulted in two new protocols: one based on locking [Lin90] and the other on timestamp ordering. In the area of experimental database systems, we have been developing a suite of database systems on several platforms. Currently, our research utilizes the UNIX, StarLite [Cook90], and ARTS operating systems [Tok89]. Experimental database systems we have developed on these platforms are the Multi-user Real-time Database (MRDB), Parallel Real-time Database (PRDB), and Real Time Database (RTDB), respectively [Son91]. All three systems are based on the relational paradigm. Much of our development consists of implementing new functionality on the most appropriate platform, and where applicable, porting the result to one of the others. In this paper, we outline the scheduling protocol based on timestamp ordering and our experience with PRDB development.

2. An Optimistic Concurrency Control for Real-Time Transaction Scheduling

In real-time transaction scheduling, the actual execution order of operations is determined by two factors: priority order and serialization order among transactions in system. The difficulties in real-time transaction scheduling arise from the fact that these two factors have different natures and are constructed in different ways. While serializable execution order is strictly bound to the past execution history, the priority order does not reflect the past execution history and may dynamically destroy the order set up in the past execution, hence serializability. By identifying the effects of the interactions between serialization order and priority order in scheduling real-time transactions, we can build more intelligent conflict resolution schedulers.

One approach to real-time transaction scheduling is to make the priority order and serialization order compatible as much as possible in order to increase the probability of satisfying both timing and

This work was supported in part by ONR contract # N00014-88-K-0245, by NOSC, and by IBM FSD.

consistency constraints. One way to make the two orders compatible is to adjust serialization order dynamically to priority order. This approach can be justified because serialization order is not subject to timing constraints as long as it enforces serializability, while we assume that the priority order of a transaction is statically determined when it arrives in the system.

Integrating a concurrency control protocol with priority-based scheduling methods has the inherent disadvantage of being limited by the concurrency control protocol on which it depends. Two-phase locking and timestamp ordering depend on the immediate validation of operations, and do not provide a facility to adjust serialization order dynamically to priority order. To adjust the serialization order, we need to delay determining the serialization order of conflicting operations, because once the serialization order is determined, the orders of operations from transactions cannot be adjusted dynamically.

In optimistic concurrency control in which the serializability test (called the *validation test*) is made only at the end of a transaction, the serialization order can be constructed dynamically in compliance with transaction timeliness and criticality. Furthermore, owing to its potential for a high degree of parallelism, optimistic concurrency control is expected to perform better than two-phase locking or timestamp ordering in real-time transaction scheduling.

We have developed an optimistic concurrency control protocol based on the notion of dynamic timestamp allocation [Bok87]. In this protocol, the serialization order is dynamically constructed by using intervals of timestamps. The protocol uses a *backward validation* scheme, in which validating a transaction is performed against committed transactions. It also updates the timestamp intervals of active transactions to adjust their serialization order. As in other optimistic protocols, the execution of a transaction in our protocol is divided into three phases: read, validation, and write. However, unlike other optimistic protocols, conflicts and nonserializable executions are detected during the read phase of transaction execution, minimizing wasted work due to later restarts of transactions.

The goal of this protocol is to enforce serializability by satisfying the following two conditions (C1) and (C2) through every read, prewrite, and validation. As long as (C1) and (C2) are satisfied, serialization order can be adjusted in favor of priority order without violating data consistency.

(C1) Each timestamp interval constructed when a transaction accesses a data object should preserve the order induced by the timestamps of all committed transactions which have accessed that data object.

(C1) The order induced by timestamp values of a validating transaction should not destroy the serialization order constructed by the past execution, i.e., by committed transactions.

Before describing the algorithms for the read and validation phases, we summarize the information used to keep track of the dependencies among transactions:

- for each active transaction T, its readset, $RS(T)$, and writeset, $WS(T)$;
- for each committed transaction T, a timestamp $ts(T)$ assigned in its validation phase;
- for each active transaction T and for each data object x it has read or written in its read phase, an interval of timestamps $I(T,x)$; and
- for each data object x, $RTS(x)$ and $WTS(x)$, which denote the largest timestamps of the committed transactions having read or written x, respectively.

In order to decide whether a transaction T is involved in a nonserializable execution, all the timestamp intervals of T are grouped as $I(T) = \bigcap_{x \in X} I(T,x)$ for X being the set of data objects accessed by T. $I(T)$ preserves the order between T and committed transactions. Any operation of an active transaction T which introduces a nonserializable execution can be detected by checking whether the execution of the operation results in $I(T) = \emptyset$.

In the implementation, with each transaction T is associated its *current interval* $I_c(T)$ instead of $I(T,x)$'s and $I(T)$. At the start of T, $I_c(T)$ is initialized as $[0, \infty)$ (the whole set of allowable timestamps). For each read or prewrite made by T, $I_c(T)$ is adjusted according to dependencies induced by the operation to satisfy (C1). A transaction T must be restarted when $I_c(T) = \emptyset$. The gradual construction of a serialization order by using $I_c(T)$ makes it possible to detect nonserializable executions even before the transaction reaches its validation phase. Furthermore, every transaction that reaches its validation phase is guaranteed to commit in this protocol.

We present the protocol via the following pseudo code. We bracket a critical section by "<" and ">", and assume that timestamp intervals contain only integers.

Read phase

< **for** every data object x in $RS(T_i)$ **do**
  $I_c(T_i) := I_c(T_i) \cap [WTS(x)+1, \infty)$ >
  if $I_c(T_i)=\emptyset$ **then** restart(T_i)

< **for** every data object x in $WS(T_i)$ **do**
  $I_c(T_i) := I_c(T_i) \cap [WTS(x)+1, \infty) \cap [RTS(x)+1, \infty)$ >
  if $I_c(T_i)=\emptyset$ **then** restart(T_i)

Validation and Write phase

< choose $ts(T_i)$ in $I_c(T_i)$
  update $RTS(x)$ and $WTS(x)$ for every x in $RS(T_i)$ and $WS(T_i)$
  adjust $I_c(T_j)$ >
  make its updates permanent in the database

The validation of a transaction means that the execution of the operations from the transaction is serializable, and the execution should be reflected in the serialization order of committed transactions. Thus we should choose a timestamp for the transaction to satisfy (C2), update *RTS* and *WTS* for data objects it accessed, if necessary, and adjust the timestamp intervals of all active transactions which conflict with it to satisfy (C1). Any timestamp $ts \in I_c(T_i)$ satisfies the condition (C2). The adjustment procedure is as the following:

Interval Adjustment Operation

< **for** every data object x in $RS(T_i)$ **do**
  for every transaction T_j which has written x **do**
    $I_c(T_j) := I_c(T_j) \cap [ts(T_i)+1, \infty)$ >
    if $I_c(T_j)=\emptyset$ **then** restart(T_j)

< **for** every data object x in $WS(T_i)$ **do**
  for every transaction T_j which has read x **do**
    $I_c(T_j) := I_c(T_j) \cap [0, ts(T_i)-1]$
  for every transaction T_j which has written x **do**
    $I_c(T_j) := I_c(T_j) \cap [ts(T_i)+1, \infty)$ >
    if $I_c(T_j)=\emptyset$ **then** restart(T_j)

The Adjust procedure given above can be modified in several ways to integrate priority scheduling with this protocol. As a simple approach, we can adjust the size of $I_c(T_j)$ of an active transaction T_j. Because the size is correlated with the probability of restarting of the transaction, for priority scheduling, a transaction with higher priority needs to have a larger timestamp interval than a transaction with lower priority. When adjusting the timestamp intervals of active transactions, if we give larger timestamp

intervals to transactions with higher priority over transactions with lower priority, then we can decrease the risk of restarting higher priority transactions. The choice of a timestamp of the validating transaction also has a definite effect on the active transactions which conflict with it, because the timestamp intervals of those transactions are adjusted according to the timestamp chosen.

As another approach, the *priority wait* strategy [Har90] in which the validating transaction waits for the conflicting transactions with higher priority to complete, can also be used in this protocol. The advantage of this strategy is that a higher priority transaction is not restarted due to the validation of a lower priority transaction. While a lower priority transaction is waiting, it is possible that it will be restarted due to the validation of one of the conflicting higher priority transactions.

3. A New Parallel Paradigm for Real-Time Database System

One important advance in computing technology is the emergence of parallel computers. In a database system, there are at least two levels in which parallelism can be exploited. The first level contains the basic database operations. The basic idea behind these algorithms is to partition a single database operation into multiple sub-operations, perform those sub-operations simultaneously and then combine the separate results into one. For example, the join operation can be performed in parallel by dividing one of the two relations into several blocks and joining each block with the other relation simultaneously. As a large amount of data are usually involved in each database operation, it is essential from a performance standpoint that accessing the data should be done efficiently. New techniques to organize indices and to structure data files are needed.

The second level is the query processing level in which different queries can be executed simultaneously if they do not conflict. For example, two CREATE operations can be executed in parallel on different processors or the interpretation of two expressions can be done simultaneously. Here we are only concerned with parallelism at the second level.

PRDB is an experimental, real-time database system that runs on an emulated tightly-coupled, shared-memory multiprocessor system in the StarLite software development environment, running on UNIX under SunView/X Windows. The overall design goal of PRDB is to provide a general paradigm for exploring parallelism and implementing different real-time scheduling policies in database systems. The paradigm has evolved from the WorkCrew model [Rob89]. The major advantage of the WorkCrew paradigm is its efficient mechanisms to control and manage parallelism by creating the minimum number of processes in the system and the employment of a lazy evaluation technique for posted work. The synchronization of concurrent tasks and the overhead of task decomposition are minimized.

In the WorkCrew paradigm, tasks are assigned to a finite set of workers. A task may consist of several subtasks. If some of the subtasks can be executed in parallel, they are put into a "request_help" queue of the worker. Any idle worker can take over the subtasks and execute them. The WorkCrew paradigm has two advantages. First, much of the work associated with task division can be deferred until a new worker actually undertakes the subtask, and avoided altogether if the original worker ends up executing the subtask serially. Second, the number of active workers in the system is always equal to the number of processors.

However, the WorkCrew paradigm has two limitations that prevent it from becoming a general framework for parallel computing. The first limitation is that there is no general mechanism to retrieve results. In the WorkCrew model, the results of operations are reflected in the preallocated space. If operations produce some new results apart from the results stored in preallocated space, which is usually the case for most of the applications, there is no way to retrieve those results. The second limitation is that there is no way to specify different operations to be performed on data, i.e., the procedure to manipulate a set of data cannot be explicitly passed to each worker so that the worker can perform different operations. Further, the WorkCrew model does not address the real-time requirements of the application.

In our paradigm, the first limitation is addressed by providing a result queue for the crew. The second limitation is dealt with by passing the handler for operations as a parameter to each worker. These

improvements require the extension of the concept of work. The concept of work in the WorkCrew paradigm is a passive entity and consists only of the data items to be manipulated. In the PRDB paradigm, the concept of work is still a passive entity, however, the contents of work not only consist of data items to be manipulated, but also the operation to be performed on the data items and the timing-constraint information for the work to be performed.

The real-time transaction scheduler and the CPU schedulers (called *dispatchers*) are separated. The real-time transaction scheduler is implemented by the crew, while the dispatcher is implemented within each worker. The real-time transaction scheduler schedules tasks according to its own policies and puts them onto two work queues residing on the crew. One of these two queues is for hard deadline tasks and the other is for soft deadline tasks. Since each worker has also its own "request_help" queue, the search path of work to do by an idle worker begins with the hard-deadline queue of the crew, then the "request_help" queues of the workers, and finally the soft deadline queue of the crew. If the deadline has passed, the workers immediately write the result into the result queue indicating the missing of a deadline. Otherwise, the work is performed and results are returned through the result queue. In the case where a worker has to synchronize with other workers in performing a task, the worker blocks and a new worker is created to help the other workers' work. Thus, the number of the active workers is always equal to that of the processors in the system, if the work load is high.

The data structures of a unit of work and a unit of result are as follows:

```
WORK = RECORD
            critical : CARDINAL; (* hard vs soft deadline *)
            deadline : Time; (* the deadline is checked before executing the operation *)
            operation : PROCEDURE; (* specifying the operation *)
            paramAddr : ADDRESS; (* pointer to the work to be done *)
            size : CARDINAL; (* the size of the work data structure *)
END;
RESULT = RECORD
            missDeadline : BOOLEAN; (* missed deadline? *)
            finishTime : Time; (* the finished time of a unit of work *)
            resultAddr : ADDRESS; (* pointer to the result data structure *)
            size : CARDINAL; (* the size of the result data structure *)
END;
```

The major functions provided by the paradigm are starting a crew of workers, destroying a crew of workers, modifying the number of workers in a crew, assigning work to a crew, requesting help by a worker, testing whether the requested work has been done by other workers, and waiting for some work to be finished.

Each basic database operation is written by using the functions provided above if some part of the basic database operation can be done in parallel. Initial results have indicated the soundness of the paradigm for parallel real-time database computing. More thorough experiments are being carried out. We believe that this new paradigm will scale well to large number of processors in the system and will be efficient in scheduling real-time transactions.

The data given below are the relative speedups of PRDB over the RDB system. The workload for the experiments is the same for the uniprocessor which runs the RDB system and the multiprocessor system which runs PRDB. The first experiment (Test1) consists of 26 "Create" operations and 22 "Insert" operations. Each "Insert" operation inserts 15 tuples in a different relation with three attributes each. Other experiments (Tests 2 and 3) consist of the same operations as Test1, however, each "Insert" operation in Test2 inserts 25 tuples, while each "Insert" operation in Test3 consists of 50 Tuples. The results show that PRDB favors coarse-grained parallelism in the computation.

Speedup of PRDB over RDB							
Number of processors		1	2	3	4	5	6
Test1	Time Units	4613	3704	3074	2593	2515	2447
	Speedup		1.24	1.50	1.77	1.83	1.88
Test2	Time Units	9046	5761	4170	3471	3120	2904
	Speedup		1.57	2.16	2.60	2.89	3.11
Test3	Time Units	26195	14276	9878	7813	6752	5841
	Speedup		1.83	2.65	3.35	3.87	4.48

4. Concluding Remarks

A real-time database manager is one of the critical components of a real-time system. To satisfy timing requirement, transactions must be scheduled considering not only the consistency constraints but also their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. In this paper, we have presented new paradigms that exploit the ideas of dynamic adjustment of serialization order and parallel computing. We are currently working on the performance evaluation of new paradigms using the prototyping environment as well as experimental database systems.

REFERENCES

[Bok87] C. Boksenbaum, M. Cart, J. Ferrie, and J. Pons, "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987.

[Cook90] R. Cook, and Y. Oh, "The StarLite Project," *The 3rd Sym. on Frontiers of Massively Parallel Computation*, Univ. of Maryland,College Park, Oct.1990.

[Har90] J.R. Haritsa, M.J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Lin90] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Rob89] E. S. Roberts, and M. T. Vandevoorde, "WorkCrews: An Abstraction for Controlling Parallelism," *DEC SRC Technical Report*, April 1989.

[Son90] S. H. Son, "Real-Time Database Systems: A New Challenge," *Data Engineering*, vol. 13, no. 4, Special Issue on Directions for Future Database Research and Development, December 1990.

[Son91] S. H. Son, M. Poris, and C. Iannacone, "Implementing a Distributed Real-Time Database Manager," *The Second International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991.

[Tok89] H. Tokuda and C. Mercer, ARTS: A Distributed Real-Time Kernel, ACM Operating Systems Review, 23 (3), July 1989.

Generating Synthetic Workloads for Real-Time Systems

Daniel L. Kiskis and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

ABSTRACT

In this paper, we describe a software system which generates synthetic workloads for use in the performance evaluation of distributed real-time computer systems. The software system consists of a high-level description language and its compiler. The language provides a flexible, easy-to-use description of the structure and behavior of the real-time workload. The compiler, called a synthetic workload generator (SWG), uses this description to produce an executable synthetic workload (SW). The SW may then be used to drive the system under evaluation while measurements are being made.

1 Introduction

Real-time systems have strict performance requirements. To determine if these requirements are met, the performance of a system is evaluated through experimentation. During the experiments, the values of selected performance indices are measured while the system is running a workload. The selection of the drive workload directly influences the results of the evaluation.

One possibility in the selection of the drive workload is to use the actual application software. However, there are a number of situations where the real workload is unavailable or unrealistic. Such situations include new systems where an application workload has not yet been developed and critical systems where, for safety reasons, performance evaluations must be done off-line. In these cases, we advocate the use of an SW as the drive workload. An SW consists primarily of a set of parameterized synthetic application tasks (SATs) which execute on a system and produce demands for resources. It also includes a driver task which controls the actions of the SW to facilitate the use of the SW during experimentation. It controls when the SW starts and stops. It also determines when the individual tasks execute.

In this paper, we describe a suite of software tools which we have designed and implemented to support the specification, generation, and execution of SWs for a distributed real-time system. This suite provides the high level support necessary to efficiently produce SWs which are customized for a particular evaluation. The suite consists of the synthetic workload generator (SWG) and some minor support programs. The SWG compiles a description of the workload

The work reported in this report was supported in part by the NASA under Grant No. NAG-1-296 and NAG-1-492 and the Office of Naval Research under Contract No. N00014-85-K-0122.

that is specified in the synthetic workload specification language (SWSL). SWSL describes the structure of the SW based on a dataflow model.

There are two primary goals in the design of the SWG suite. The first is to be capable of accurately representing actual real-time workloads. This goal is met through the selection of an appropriate workload model. The model was chosen to reflect the structure of the software which composes the workload being modeled. By accurately modeling the structure of the workload, we also capture many of its behavioral characteristics. Representativeness is enhanced by the selection of parameters for the objects in the workload. Parameters are defined for both the SATs and the resources that they use and, possibly, share. These parameters were selected to reflect both common software properties and those properties which are specific to real-time software.

The second goal in the design of the SWG suite is ease of use. All components of the suite should be easy to use while retaining their flexibility and power. Ease of use is enhanced by the simple, regular structure of SWSL. The language structures allow one to change both the values of parameters and the interactions between SATs with little effort. We also provide a simple user interface to the SWG. It is completely automated to handle all the various compilation stages and their corresponding intermediate files.

This paper is organized as follows. In Section 2 we describe the notation used to specify the SW. In Section 3 we discuss the functions of the SWG, and in Section 4 we discuss the SW which is being supported by the SWG suite. Section 5 we give our summary and discuss our future work.

2 The Workload Model

The workload model provides a high level description of the structure of the workload. We represent this structure using a dataflow notation. A dataflow notation was chosen because it is commonly used to specify software structure. Workload specifications in other dataflow notations may be easily translated into our dataflow notation. The translated workload will retain the structure of the original. Hence, it will be quite representative. Using our notation, we can specify both the individual tasks and the interactions between tasks.

2.1 Task Level Notation

The notation is divided into two levels of abstraction. The higher level, or *task level*, defines the tasks, the resources they use, and their interactions. The task level notation uses formalisms borrowed from the area of structured analysis (SA). In particular, we base the notation on ESML, an SA notation created by Bruyn *et al.* [2]. ESML was developed for the high-level specification of real-time software. It is a combination of the Ward/Mellor [10, 9] and Boeing/Hatley [4] SA notations. These two notations were independently derived and use differing approaches to add timing and control information to the basic data flow model developed by DeMarco [3].

By basing our notation on the SA notation, we accomplish our primary goals. First, we tie the structure of the SW directly to the structure of the workload, thus improving the ability of the notation to accurately model actual workloads. Our notation is the first to be based on a high-level software specification notation. Previous systems were based either on low-level

specifications such as flowcharts [1, 8] or on high-level notations such as UCLA graphs [6] which are not related to software specification notations. Second, we make it easier for the user of the SWG suite to produce workloads. The SA notations are commonly used by CASE tools for high-level software specification. Hence, it is likely that the actual or proposed workload being modeled has been specified in terms of an SA or similar notation. To produce a description of the workload in our notation, the user must translate the specifications. This process may be performed manually, or may be automated as part of a CASE tool. By using a similar notation, we simplify the translation.

SWSL defines the workload in terms of transformations, flows, stores, and terminators. Transformations represent units of computation, generally tasks. Flows are data and control paths. Stores are units of data storage, and terminators are interfaces between the workload and the environment. The parameters for these objects define characteristics such as task interactions, scheduling requirements for tasks, and access properties of shared objects.

2.2 Operation Level Notation

The lower level abstraction in the notation is the *operation level.* It defines the task's internal structure, behavior, and the manner in which it uses resources. This notation is similar to that used by Singh and Segall [7] in the Pegasus system. A task is defined in terms of sequences of operations and control logic. Each operation represents the use of a single resource by the task, and the control logic determines the sequence in which the resources are used.

The control structures consist of loops and branches. They execute probabilistically to simulate the variation of program execution based on the value of the task's input data. Hence, the SATs simulate the random execution time distributions of real application tasks. The control structures also cause the SATs to simulate the resource usage patterns of the real application tasks and not just the quantities of resources used. By simulating the random execution times and the resource usage patterns, the SW models the workload more realistically. This realism is necessary when studying real-time systems. The SW must express the time-specific behavior of the workload. It is this behavior which affects the real-time aspects of the system.

3 The Synthetic Workload Generator

The SWG compiles the SWSL specification to produce the SW. It reads the task level description and produces parameter tables. These tables describe the structure and parameters of the task level notation in a form that may be used by the SW driver. The SWG compiles the operation level description to produce C code. Each operation in the description is expanded into its equivalent code. This code is stored in a library containing code for all possible operations. Later, the SWG invokes the C compiler to create the object code for the SATs. This object code is then linked with the parameter tables and the object code for the SW driver to produce the complete executable SW.

The SWG offers a number of support features to aid in the creation of SWs. It performs syntax and semantic error handling on the input files. It also does consistency checking on the dataflow graph for the workload. It enforces the construction rules for the notation, thus reducing the probability of logical errors in the SW.

The SWG provides another important feature. It supports the automatic creation of replicated objects from templates in the SWSL specification. This feature is used when multiple tasks in the workload have the same parameter values. The user specifies the structure of one instance of the task. The task definition states that a copy of the task be executed on each of a number of different processors. Those copies are then generated automatically. The user does not need to individually program the specifications of each copy of the task. The SW specifications are therefore smaller and less likely to contain errors.

Replicating tasks involves both creating copies of the task and resolving naming conflicts caused by the replication. Copying the task is simple; resolving the naming conflicts is more difficult. Name resolution involves processing each task in the workload. Any reference to the replicated task must be replaced with a reference to the appropriate copy of the task. SWSL defines rules for determining which copy to reference. These rules may be superseded in the specification of an individual component by explicitly specifying which copy is to be used.

4 The Synthetic Workload

The output of the SWG is an executable SW. Our prototype SW is described in [5]. The SW executes on a distributed system. Each processor executes a driver task and the appropriate SATs. The driver controls the activities of the SW in the context of the experiment. An experiment is divided into a number of independent runs. A *run* is a single execution cycle of the SW. During each run, the SW is initialized by the driver, and the SATs execute and eventually terminate. In the SWSL description of the SW, the user specifies the number of runs. For each component of the workload, different parameter values may be specified for each run.

At the beginning of each run, the driver initializes the SW. It reads the parameter tables which were produced by the SWG and creates the specified SATs. Next, the drivers on all processors synchronize. Once synchronized, they begin the execution of their respective SATs as specified by the SATs' parameters for that run. By synchronizing at the beginning of each run, the driver ensures that the SW's behavior will stabilize quickly. The SW must be executing stably before accurate measurements may be made on the system. There are two ways to specify the end of a run in the SWSL specification. The first is to specify a time limit for the run. The second is to specify a condition, which, when met, indicates to the driver that the run has completed. An example of such a condition is the completion of N executions of a specific periodic SAT. When the driver determines that a run is over, it stops the execution of the SW. All SATs are reset and system resources are returned to their initial states. The driver then waits before beginning the next run. This wait gives the user an opportunity to upload locally stored performance data or to reset external measurement devices. The driver begins the next run when it receives a signal from the user.

The SW is designed to be compatible with a wide range of performance measurement techniques. It executes as an application on the target system. Therefore, it may be used with any measurement mechanism which is part of the hardware, system software, or which is external to the system. It requires no special support and therefore will not interfere with these mechanisms. It also may be used with software measurement mechanisms which are not part of the system software. These measurement tasks may be specified as SATs. They will be invoked by the driver and will execute for the duration of the run.

5 Summary and Future Work

As real-time systems become larger and more complex, we need more sophisticated tools to analyze their performance. The SWG suite is one such tool. It is designed to produce SWs which execute on distributed real-time systems. The workload model and corresponding language are specifically defined to describe the structure and behavior parameters of real-time workloads. The SWG supports features such as replication of tasks which facilitate its use on a distributed system. Finally, the SW is designed to support experimentation.

The SWG as described is operational. All functions described in this paper have been implemented. We will be using the SWG to make baseline performance measurements of the experimental, distributed real-time system HARTS and its operating system HARTOS. Both HARTS and HARTOS are under development at the Real-Time Computing Laboratory at the University of Michigan. As we use the SWG suite and become more experienced with the problems of performance evaluation, we will be upgrading the SWG software to incorporate new features.

References

[1] R. Baird, "APET - a versitile tool for estimating computer application performance," *Software - Practice and Experience*, vol. 3, pp. 385–395, 1973.

[2] W. Bruyn, R. Jensen, D. Keskar, and P. Ward, "ESML: An extended systems modeling language based on the data flow diagram," *ACM Software Engineering Notes*, vol. 13, no. 1, pp. 58–67, 1988.

[3] T. DeMarco, *Structured Analysis and System Specification*, Prentice-Hall, New Jersey, 1978.

[4] D. J. Hatley and I. A. Pribhai, *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, 1987.

[5] D. L. Kiskis and K. G. Shin, "A synthetic workload for real-time systems," in *Proc. Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pp. 77–81, May 1990.

[6] A. Singh, *Pegasus: A Controllable, Interactive, Workload Generator for Multiprocessors*, Master's thesis, Carnegie-Mellon University, December 1981.

[7] A. Singh and Z. Segall, "Synthetic workload generation for experimentation with multiprocessors," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 778–785, 1982.

[8] R. E. Walters, "Benchmark techniques: a constructive approach," *The Computer Journal*, vol. 19, no. 1, pp. 50–55, February 1976.

[9] P. T. Ward, "The transformation schema: An extension of the data flow diagram to represent control and timing," *IEEE Trans. Software Engineering*, vol. SE-12, no. 2, pp. 198–210, February 1986.

[10] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time System*, volume 1-3, Yourdon Press, Englewood Cliffs, 1986.

Managing Beliefs, Desires, and Time in Real-Time Systems

Tom Bihari[†] Prabha Gopinath[‡] Tom Walliser[†]

[†]amt@eagle.eng.ohio-state.edu
Adaptive Machine Technologies
1218 Kinnear Road
Columbus
OH 43212

[‡] psg@philabs.philips.com
Philips Laboratories
North American Philips Corp.
345 Scarborough Road
Briarcliff Manor, NY 10510

January 11, 1991

1 Introduction

Interest has been increasing in intelligent hard-real-time systems. This interest is driven by applications such as the Pilot's Associate [3] and others, which require high-level reasoning abilities within a hard-real-time environment. Tackling these applications requires combining hard-real-time technology with intelligent-system technology. Much of the existing research in *hard-real-time systems* (HRTS) has been directed at specifying, expressing, and fulfilling the timing requirements of a system. Such efforts have been oriented towards the concrete design-implementation models [5] of a system (e.g., processes and deadlines). Timing constraints for a system are usually determined by the interactions of the system with the external environment. Existing hard-real-time systems are designed to satisfy real-time constraints in applications where timing errors can have serious consequences. However, such systems are not usually required to exhibit intelligent "reasoning" behavior. A typical example of such a hard-real-time system is a control computer for a dynamically unstable, fly-by-wire aircraft.

In contrast, much of the research in *intelligent real-time systems* (IRTS) has been directed at *reasoning about* time with respect to abstract models of a system (e.g., tasks, environments, and intelligent agents [6]). Such reasoning may include *choosing* timing constraints based on the goals of the system. Existing prototype systems exhibit intelligent behavior and some degree of real-time behavior. However, such systems are not usually driven by hard-real-time constraints. Typical examples of intelligent real-time systems include wheeled robots which are required to negotiate a room containing obstacles, using ultrasonic or vision sensors. The robots are usually permitted to stop and "think" as necessary.

Lately, however, driven by advances in both HRTS and IRTS technology, and by the increasing complexity of potential applications, the two areas of interest are beginning to converge. We believe that there is a need for design methodologies that are consistent with both areas and that can be used for designing well-integrated systems.

Existing systems, such as the Adaptive Suspension Vehicle [2], which combine the need for high-level reasoning with the need for hard-real-time performance, are often organized in distinct layers, where the higher-level layers are responsible for functions, such as planning, which are typically associated with the IRTS domain. These layers typically have flexible timing constraints. However, as one moves down the layers of the system, the functionality of the layers shifts towards HRTS technology, with greater emphasis being placed on meeting deadlines and other time-related specifications.

In this paper we describe an object-oriented programming methodology, and examine its appropriateness as a model for combining the HRTS and IRTS technology. As a specific example, we discuss the notions of *desires* and *beliefs* taken from IRTS, and the implications of their application to the HRTS problem of controlling a robot arm. We propose that complex applications, such as these, are best represented as a hierarchy of objects, where the desires and beliefs of objects are represented as object attributes.

2 Agents, Beliefs, Desires, and Goals

Shoham [6] describes *agent-oriented programming* as a computational framework in which *agents* - intelligent objects - relate to one another, and the environment, based on such concepts as *beliefs*, *desires*, and *goals*. Agents believe

propositions about the world. Agents have desires about the world. When an agent decides to actively pursue a desire, it formulates an appropriate goal. In this paper, we will equate desires with goals for simplicity. We assume that if an agent desires something, it immediately forms a corresponding goal. In Shoham's language, for example,

$$< 9:15, B_{Harold} < 9:30. G_{Calvin} < 10:00, weld(Calvin, Car14) >>>$$

means that at 9 : 15, Harold believes that at 9 : 30, Calvin will have the goal of welding Car14 at 10 : 00. When parsed in an LR fashion, such a statement can be viewed as being a hierarchy of beliefs and goals. However, at any point during the execution of the application, there may exist a discrepancy between the beliefs (goals) of one layer and the goals (beliefs) of an immediately neighboring layer. Such discrepancies result from race-conditions, timing-skews, or simply delays in propagating information through the hierarchy. Therefore, when such high-level statements are translated into lower-level constructs, they must be augmented with timing constructs which capture the acceptable granularity of belief (goal) discrepancy.

Agents communicate by sending messages with well defined semantics. Two important message classes are *Inform* messages and *Request* messages. Inform messages transfer beliefs and goals between agents. Request messages allow agents to ask that other agents perform actions.

Shoham's research objectives include the automatic translation of such high-level statements into language constructs that can directly control the associated machines. Our goals are somewhat similar. We believe that, as intelligent real-time systems become more complex, it will become necessary to continually trade the timeliness of an action for the quality of its result. To do so in actual operation, it will be necessary to encode the meanings of timeliness and quality in a form that can be accessed by both human and automated decision-makers (e.g., processor schedulers).

3 Timeliness

In real-time applications, time constraints are derived from the entities in the real world (e.g., agents, environments and tasks) and the relationships among them. In robotics, for example:

- Servo-control time constraints derive from the structures and dynamics (e.g., natural frequencies) of the objects being controlled (e.g., the aluminum links of a robot manipulator), and how they interact.
- Elemental-move motion-planning (i.e., planning how to move from point A to point B) time constraints derive from the position and velocity required of the object and the positions and velocities exhibited by other objects (obstacles).
- High-level, task-planning time constraints derive from the overall capabilities of the agent (e.g., a robot firefighter), the characteristics of the environment (e.g., a fire site), and the requirements of the task (e.g., putting out the fire).

The flexibility of these time constraints (hard deadlines vs soft deadlines), and the degree to which they can be traded for various levels of quality of the result, depend on the characteristics of the real world from which they are derived. For example, it is difficult to dynamically change the natural frequency of a physical robot arm, so the arm's servo controller must operate near a given frequency. Fortunately, at the servo level, the arm's world is fairly static and well understood. The servo computations usually take a fixed or tightly bounded amount of time. In contrast, the task planner for a firefighting robot is operating in a more dynamic and less understood world. It is not possible to plan optimally, using any fixed amount of computing resources, since there is always too much information, some of which is wrong, and some of which will become outdated by new information. Fortunately, the task planner may be able to generate acceptable sub-optimal plans, as time and resources allow.

4 The Object-Oriented Model

Our model for integrating HRTS concepts and design with IRTS concepts and design is built around our prior work on hierarchical interacting objects [1, 4]. Objects typically correspond to real-world entities (e.g., manipulators, sensors, etc.), although this is not strictly necessary. Figure 1 shows a system containing two objects: a Robot Arm which can move to different positions and grasp payloads, and a Planner which generates task plans for the Arm to perform.

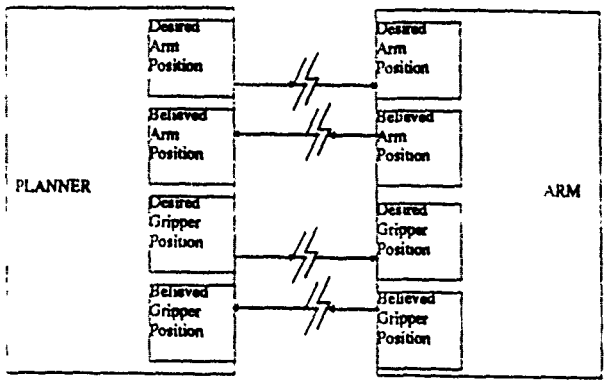


Figure 1: A Real-Time System

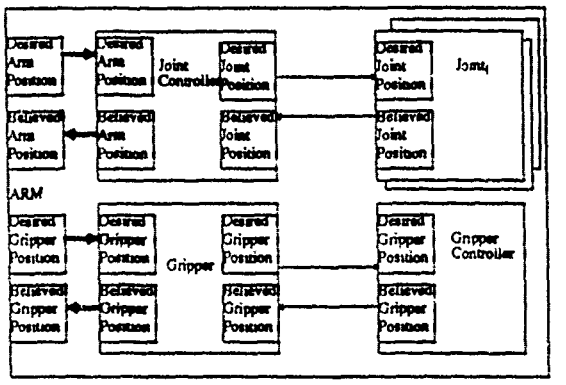


Figure 2: The Arm Object and Its Sub-Objects

Objects have attributes which have values. These values typically represent some aspect of the associated entity (e.g., the velocity of a robot arm) represented by the object. Specifically, attributes can represent the beliefs and desires of the objects. Attribute values are tuples with the form

$$< GenerationTime, ValidTime, Value >$$

The *ValidTime* is the time at which the *Value* is supposed to correspond to the state of the associated real-world entity. The *GenerationTime* is the time at which the tuple was created. This creates a two-dimensional space of values for each attribute. Values themselves may be complex entities, containing, for example, the expected value, tolerances, probabilities, and so on [1].

Objects are typically composed of sub-objects. Figure 2 shows the sub-objects of the Arm in Figure 1. The attributes of such a composite object may be synthesized from those of its sub-objects (e.g., mass), or they may reflect features unique to the composite object which are not found in the sub-objects (e.g., An Arm can "lift" a payload, but a Gripper cannot.).

Objects interact by passing values among their attributes. Consider the Planner-Arm system. At any instant, the Planner's desires and beliefs concerning the Arm may be separate from the Arm's desires and beliefs. For example, the Planner and Arm may have the following attribute values:

Object	Attribute	Value
Planner	BelievedArmPosition	< 12:00, 1:00, 111 >
Planner	DesiredArmPosition	< 12:00, 1:00, 222 >
Arm	BelievedArmPosition	< 12:00, 1:00, 333 >
Arm	DesiredArmPosition	< 12:00, 1:00, NONE >

This says that, at 12:00, the Planner believes that the Arm's 1:00 position will be 111, but desires that it be 222. At 12:00, the Arm believes that its 1:00 position will be 333, and it has no desired position.

[1] The attribute boxes in Figure 1 should not be confused with single buffers which are written and read. They simply represent the objects' desires and beliefs. In actual implementation, these boxes might contain multiple buffers and queues.

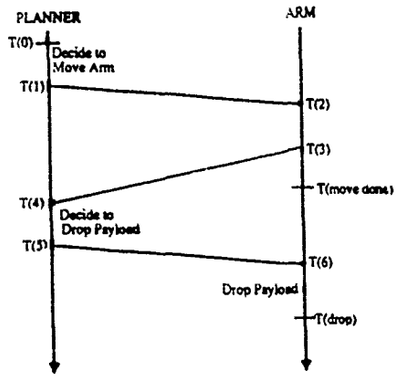


Figure 3: A Real-Time System

In a real-time system, objects frequently operate in *Choose* → *Execute* → *Evaluate* cycles. Based on its existing beliefs, an object chooses a plan of action. It executes the action (by informing other objects of its desires), then evaluates the results (by updating its own beliefs). In the Planner-Arm system, this cycle might include the transmission of an Inform message containing the value of the Arm's Believed-Arm-Position to the Planner's Believed-Arm-Position and transmission of an Inform message containing the value of the Planner's Desired-Arm-Position to the Arm's Desired-Arm-Position.

5 An Example

In this section, we will describe an example application, specify its required behavior, analyze the information content and flow, and discuss design and implementation alternatives.

5.1 Task Specification

Consider the system in Figures 1 and 2. Suppose that in this application, the Arm is required to ungrasp (and drop) a payload, which it is already holding, at a specific position P(drop), at a specific time T(drop). If the Arm is not in position P(drop) at time T(drop), the payload should not be dropped.

The task plan generated by the Planner consists of two statements:

1. The Planner desires that the Arm be in position P(drop) at time T(drop).
2. IF the Planner believes that the Arm will be in position P(drop) at time T(drop), THEN the Planner desires that the Arm ungrasp the payload at time T(drop).

5.2 Information Analysis

Figure 3 shows a possible sequence of events for the task. Based on its beliefs at time T(0), the Planner decides to begin the two-step plan. At T(1), the Planner informs the Arm that it desires the Arm to be in position P(drop) at time T(drop). At T(2), the Arm begins the activities necessary to move the Arm. This may include iterations of the Arm's own *Choose* → *Execute* → *Evaluate*, made up of small-grain motion planning, sensing, and servo control. The Arm actually finishes moving to P(drop) at T(move done). This time may vary, depending on the previous position of the Arm.

The Planner must know the likely success or failure of the Arm's movement by T(4), if the Planner is to request that the Arm open its gripper by T(drop). Therefore, the Arm must inform the Planner of its expected position at T(drop) no later than T(3).

The critical consideration for the belief accuracy of this system is the relationship between T(3) and T(move done). If T(3) is before T(move done), then at T(3) the Arm cannot know for certain that its move will be successful. It must send its current belief to the Planner. Suppose the Arm informs the Planner that it anticipates a successful move. Between T(3) and T(move done), the Arm may encounter an obstacle or other problem preventing its move to P(drop) by T(drop). The Arm no longer believes that it will be at P(drop) at T(drop), but the Planner still does, and acts accordingly.

Assuming that the relationship between T(3) and T(move done) varies, it may be necessary to trade the timeliness of the task (i.e., opening the gripper at exactly T(drop)) for the accuracy of the Planner's beliefs about the Arm's position. Several options exist:

1. The Planner may wait until it is confident that the Arm is at P(drop), but waiting may prevent the Arm from dropping the payload until after T(drop).

2. The Planner may request that the Arm drop the payload at exactly T(drop), without high confidence that the Arm is at P(drop).

3. The Planner may choose to abort the task, and not drop the payload anywhere, unless there is a high degree of confidence that the Arm is at P(drop) at T(drop).

The choice depends on the relative costs of dropping the payload on the wrong position, at the wrong time, or not dropping the payload.

5.3 Information Design

Figures 1 and 2 show the major components and interactions of the application. The desires and beliefs of the objects are encoded as attributes of the objects. For example, the statement: "At T(4), the Planner believes that the Arm Position at T(drop) will be P(drop)." is described by giving the Planner an attribute called Believed-Arm-Position, and giving the attribute the value: $< T(4), T(drop), P(drop >$ where T(4) is the Generation Time described in Section 4, T(drop) is the Valid Time. and P(drop) is the Value.

The transfer of desires and beliefs between objects via Inform messages consists of transferring attribute values. For example, the statement: "At T(1), the Planner informs the Arm that it desires the Arm Position to be P(drop) at T(drop). The Arm receives the information at T(2)." is represented as follows. The Planner's Desired-Arm-Position attribute has the value: $< T(1), T(drop), P(drop) >$. The $< T(drop), P(drop) >$ sub-tuple is transferred to the Arm's Desired-Arm-Position attribute, resulting in the value: $< T(2), T(drop), P(drop) >$.

Notice that only the $< ValidTime, Value >$ sub-tuple is transferred. This amounts to the Arm "internalizing" the desires of the Planner - it accepts the Planner's desires as a *command*. That is, the Arm records the statement: "At T(2), *the Arm* desires the Arm Position to be P(drop) at T(drop).". In general, it is possible to have arbitrarily complex attributes, such as the Arm attribute: "The Arm's belief about the Planner's desire about the Arm's Position".

However, we believe that, for most hard-real-time systems in the robotics domain, "internalizing" beliefs can be used to simplify designs and maintain acceptable real-time performance.

5.4 Design and Implementation Choices

Detailed design and implementation of the Planner-Arm system requires many decisions which may affect the timeliness and belief accuracy of the system. Passing beliefs and desires between objects involves sending and receiving Inform messages, and associated issues such as message initiation and message transmission delays.

In the Planner-Arm system, which object(s) should initiate the transfer of information? In our object model, as in others, two types of information transfer are possible:

1. `Value = Get(Object,Attribute,ValidTime)`

2. `Put(Object,Attribute,ValidTime,Value)`

The Put operation is simply an Inform message. The Get operation is actually two messages: a Request message asking for the information and an Inform message containing the information. Both operations transfer information. They differ in the control of the transfer. For example, if the Planner needs to know the Arm's believed position at T(drop), there are two options:

1. The Arm could send "Put(Planner,Believed-Arm-Position,T(drop),Position)" to the Planner each time the Arm's belief about its position at T(drop) changes. The Arm controls when the information is transferred.

2. The Planner could periodically send "Get(Arm,Believed-Arm-Position,T(drop))" to the Arm. The Planner controls when the information is transferred.

Option (1) may make sense when the Arm's belief changes significantly at well-defined points. When asked to move to P(drop) by T(drop), for example, the Arm might first determine if P(drop) is in its possible range of motion; then determine if the move from its current position to P(drop) is possible by T(drop); then execute the move. The Arm's belief may become more certain after each step, and pass the updated information to the Planner.

Option (2) may make sense when the Arm itself cannot judge the significance of its beliefs. This is true, for example, when the Planner is gathering general information about the Arm's state before it puts any plan into action.

As a rule, the mechanism for transferring beliefs and desires between objects is driven at least in part by the constraints on each object's ability to use the new information. If the Arm cannot abort a move in mid-motion, it makes little sense for the Planner to pass its new desires to the Arm. It may make more sense for the Arm to ask for the Planner's new desires only when the Arm has reached a state where it can process new information.

When transferring beliefs and desires, which object should keep track of message transmission delays? If we assume fixed or bounded transmission delays and that a global time base is available, each message can include the time at which the message was sent, and the receiver knows when the message was received. Therefore, it may make sense for the receiver of a message to record the transmission times of messages it receives. Then, if an object is expecting a response by a certain time, it can include the expected transmission delay for the response message in the deadline. For example, the Planner can tell the Arm that a response must be *sent* by T(3), because the Planner wants to receive it by T(4).

A general note: It might seem that one way to get around the problem of inconsistent beliefs in the Planner-Arm example is that the Planner tell the Arm the precondition for opening the gripper. That is, the Planner can tell the Arm "Open the gripper at T(drop) only if the position is P(drop) at T(drop)". This is just moving the problem, however. As Figure 2 shows, the Arm object is really just a shell containing several sub-objects. The Gripper-Controller could be requested to ask the Joint-Controller for the position before dropping the payload, or a new "Sub-Planner" object could be added to coordinate the Gripper-Controller and the Joint-Controller. That would just move some of the functionality (and belief inconsistency problems) of the Planner into the Arm. In fact, the original Planner-Arm system can be thought of as encapsulated in a larger "Smart-Arm" object.

6 Conclusion

Intelligent, distributed, real-time systems can never maintain perfect, consistent information about a complex and changing environment, particularly given the hard time constraints on their operation. System specification and design languages which represent the meanings of timeliness and quality and which can be mapped to hard-real-time implementations are needed.

We have proposed a particular object-oriented system model which can represent high-level concepts such as *beliefs* and *desires* and have discussed the translation of these concepts to practical designs and implementations.

Dynamically trading belief and desire consistency for timeliness may allow tasks which cannot be performed on-time and with complete confidence in the results to be carried out slightly late or with less confidence. Practical approaches to this problem are needed.

References

[1] Thomas Bihari, Prabha Gopinath, and Karsten Schwan. Object-Oriented Design of Real-Time Software. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 194–201. IEEE, 1989.

[2] Thomas Bihari, Thomas Walliser, and Mark Patterson. Controlling the Adaptive Suspension Vehicle. *Computer*, 22(6):59–65, June 1989.

[3] Jay S. Lark et al. Concepts, Methods, and Languages for Building Timely, Intelligent Systems. *Real-Time Systems: The International Journal of Time-Critical Computing*, pages 127–148, May 1990.

[4] P. Gopinath, T. Bihari, K. Schwan, and A. Gheith. Operating System Constructs for Managing Real-Time Software Complexity. In *Proceedings of the Workshop on Operating Systems for Mission Critical Computing*, pages U1–U9. ONR et al, 1989. Available as Philips Technical Note TN-89-110.

[5] Aloysius Ka-Lau Mok. The Design of Real-Time Programming Systems Based on Process Models. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, pages 5–17, Dec. 1984.

[6] Yoav Shoham. Agent Oriented Programming. Technical report, Stanford University, 1990.

Adding Problem-Solving Capabilities to Existing Real-Time Systems

C. J. Paul[1], Anurag Acharya[2], Bryan Black[1] and Jay Strosnider[1]

Strosnider@usa.ece.cmu.edu
(412) 268-6927

Abstract

This paper examines the fundamental difference between AI tasks and conventional real-time tasks, and discuss the problems posed by the integration of the two kinds of tasks on the same computing platform. We then develop an architecture to address the temporal isolation and responsiveness issues raised. We demonstrate the performance of this architecture and the directions for future research.

1 Introduction

Contemporary production-quality real-time systems provide little or no support for dynamic problem-solving. An important reason for this is these systems are geared towards handling processes with predictable runtime behaviour while the search-based nature of dynamic problem-solving renders its runtime behaviour inherently unpredictable. This unpredicatability has also been instrumental in preventing a widespread acceptance of the *Real-time AI* architectures being developed in research labs. Ideally, one would like to incorporate problem-solving capabilities into current real-time systems as seamlessly as possible.

We have developed CROPS5 (Concurrent Real-Time OPS5), an architecture for embedding problem-solving capabilities into existing real-time systems. In this paper, we present the architecture, and discuss issues in scheduling AI tasks in real-time systems.

2 Source of Execution Time Variance in AI Tasks

First, we examine the fundamental difference between conventional real-time tasks and AI tasks, and use this as the basis for developing the scheduling methodology.

Let us consider the run-time variance of conventional RT tasks. Typical real-time signal processing algorithms have little to no variance associated with their run-times. This is because, regardless of the complexity and size of most signal processing algorithms (FFT's, filters, etc.) there are generally no data dependencies which can cause the execution times to vary. The input data is simply processed in a uniform, deterministic fashion. On the other hand, control oriented, RT tasks will have data dependencies. As the system to be controled increases in complexity, the number of data dependencies will likely increase resulting in increased variances in RT tasks run-times. We now argue that for RT tasks and AI tasks of comparable complexity, the run-time variance of the AI tasks will be generally larger.

To address this issue, we pose the question, "What is the fundamental difference between conventional RT processes and AI processes?" Figure 1 illustrates a continum between conventional RT tasks and Blind Search AI tasks. On the far left, RT tasks are completely known, and there exists an explicit algorithm that transforms a given set of inputs to an appropriate output. There is no notion of search at this end of the spectrum. Any variations in run-time are associated solely with data dependencies. AI tasks also have data dependencies which cause variations in run-times, but an additional source of run-time variance is introduced due to search. As one moves to the right, either the task characteristics or their interactions with the environment are not completely known. A hueristic is now required to search the state space for an appropriate result. At the far right, there is no knowledge to direct the search resulting in a blind search. In this case, one would expect to have a large variance in run-times. As one moves back to the left, increasing knowledge may be applied to reduce the variations due to search. We thus argue that the

[1]Department of Electrical and Computer Engineering, Carnegie Mellon University, PA 15213
[2]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

difference between conventional RT tasks and AI tasks is due to the introduction search. Further, that this search process increases the variance of typical AI processes over their conventional RT counterparts of comparable complexity.

Given the large variance of AI tasks, and the fact that the worst case execution time is either too large or unknown, traditional methods for the design of real-time systems cannot be directly applied to AI tasks. An attempt to blindly apply these techniques will result in systems which are grossly underutilized. Innovative architectural and problem-solving approaches are required. This paper examines the architectural issues in adding problem-solving tasks to real-time systems, and presents a Real-Time Problem-Solving (RTPS) architecture to address some of these issues.

3 Background

In this section, we explore the impact of AI processes in real-time environments from a real-time scheduling point of view. We discuss what form of timing guarantees we can provide for AI tasks in real-time systems, and the scheduling implications of providing such guarantees.

Researchers in the real-time scheduling community have been developing different approaches to real-time task scheduling and the associated schedulability analysis. Algorithms like the rate monotonic scheduling algorithm have been gaining wide acceptance. The schedulability of real-time tasks can be determined using these algorithms. The notions of schedulability are predicated by a knowledge of worst case execution times, and periodicity.

But in the case of AI tasks, there is a large variance in execution time. The worst case execution time of such tasks is often many orders of magnitude larger than the average case execution time. In some cases, the worst case execution time is not known. From the scheduling standpoint, in cases where the run-time variance is large, the schedulable utilization of the computational resources is very low, resulting in very inefficient implementations. If we are willing to accept the inefficient implementation resulting from the worst-case analysis, then AI tasks can be directly integrated into existing real-time systems with little modification.

On the other hand, if we want to build systems with AI tasks, and still be able to guarantee a reasonable level of utilization, new paradigms are needed. In this paper, we present a combination of architectural and problem-solving methodologies that solve this problem.

3.1 Temporal Isolation

Given an AI task with a large variance, the AI task can be run as a background task with no modification to any of the existing scheduling algorithms. The only requirement is that the AI task be pre-emptable. But with the AI task running in the background, we cannot provide any guarantees on the response time of the AI task for any given instantiation.

To improve the predictability of the AI task, we may need to run it at a higher priority level. In this case, real-time tasks of a lower priority than the AI task will be affected by the execution time variance of the AI task, leading to potentially frequent and unpredictable load-shedding of the lower priority tasks. This is not acceptable. To enable the AI task to run at high priority, and still be able to guarantee the performance of lower-priority tasks requires that we place strict upper bounds on the guaranteed amount of time made available for the execution of the AI task. This is achieved by providing for a fixed upper limit on execution time of the AI task, and then continuing it later if background cycles are available. In this fashion, lower priority real-time tasks get a chance to execute. Some of these concepts are based on the Imprecise computation methodology developed by Jane Liu.

Restricting the computational time of the AI task in order to provide temporal isolation presents us with other problems. On what basis is the guaranteed upper bound selected? Obviously, the AI task must be able to produce some sort of a result within the time available, refining it if additional time is available later. This calls for problem-solving methodologies which can produce solutions of which are monotonically non-decreasing in quality.

This is one possible approach to providing temporal isolation between AI tasks and conventional real-time tasks. We are evaluating scheduling mechanisms for providing temporal isolation.

We now examine the issue of responsiveness.

3.2 Responsiveness

Like other tasks in the system, problem-solving tasks need to be adequately responsive to events in the external environment. Real-time problem-solving (RTPS) architectures should, therefore, have provision for low-latency interruptibility of the problem-solving tasks.

A complex real-time control system will be controlling a large number of subsystems and be monitoring an equally large number of sensors. Under such conditions multiple contingencies can arise, which require dynamically shifting priorities. Depending on the current operational state, it would be necessary to focus the attention of the problem-solving to particular subproblems. Therefore, multiple *streams* of problem-solving could be simultaneously active (but only one currently *in focus of attention*). There are two possible approaches to handling such situations:

- Dedicate a problem-solving task to a stream of reasoning. This would require the top level scheduler to be sophisticated enough to deal with dynamically changing system priorities for focus-of-attention. Introducing such problem-solving capability into the central scheduler would make the scheduler inefficient for scheduling regular periodic tasks. Also, this approach would introduce the unpredictability of problem-solving into the kernel of the system, adversely affecting the ability of the system to meet hard deadlines.

- Support the specification of multiple streams of problem-solving in the same problem-solving task. This would require the RTPS architecture to schedule the streams. The primary advantage of this approach would be that the unpredictable computational load associated with dynamic prioritization and the scheduling of these streams would be isolated from the top-level scheduler and encapsulated in the problem-solving task.

The second approach appears to be significantly better. Therefore, RTPS architectures should support the specification of multiple problem-solving streams within a problem-solving task and provide for scheduling between these streams.

To ensure fast reactivity and fast switch in *focus-of-attention*, it is necessary that the problem-solving task be able to switch between different streams at a rapid rate. Therefore, the RTPS architecture should provide for a low-latency switch between the problem-solving streams.

4 CROPS5: Concurrent Real-Time OPS5

We have designed the CROPS5 architecture in accordance with the broad design principles outlined above. It is based on the production system model and borrows heavily from OPS5[BFKM85] for syntax and semantics. CROPS5 is based on CParaOPS5[KTG+88], which is an OPS5 to C compiler developed at CMU. The system uses the Rete[For82] pattern match algorithm. CParaOPS5 supports match-level parallelism on parallel machine architectures. Our system extends OPS5 in the following significant ways:

Focus of Attention As opposed to the single problem-solving stream in OPS5, CROPS5 supports multiple problem-solving streams. Individual streams have disjoint sets of productions (and hence disjoint RETE nets). Each stream has a *private* working memory. In addition to the private working memory, the system also maintains the abstraction of a *global* working memory that can be matched against by productions belonging to any stream. Each stream has its own conflict set.

Fast Stream Switching: Associated with each stream is a stack of tokens that are yet to be matched and a buffer of working memory elements yet to be processed. A stream can therefore be characterized by

a *(Rete net, token-stack, wme-buffer)* triple. Fast switching between streams is achieved by switching between the corresponding tuple pointers.

The smallest unit of computation during the match process is the time taken to process a token. It has been shown that this is relatively constant and is of the order of 200-300 machine instructions. CROPS5 recognizes new data at every token processing boundary, and can perform stream switches to immediately start processing the data.

Multi-level Scheduler CROPS5 provides a scheduler for the streams. The unit of time for the scheduler is the token-processing time. Two step response is supported by permitting conflict resolution to operate after a specific number of tokens have been processed and then again when the match has been completed. Correctness issues are being investigated.

Enironment Interface The interface to the external world is through a Data Handler. The Data handler accepts input from the other tasks and interrupt service routines in the system. The data is then written to the global memory, and pointers to the new data are put in the buffers of each of the streams. When the stream executes next, the data is matched, and the stream can take action on the data.

High Performance CRCPS5 is based on CParaOPS5. Earlier versions of OPS5 were implemented in Lisp, and shared its garbage collection based memory management. Garbage collection is a problem from the real-time point of view, since it is not possible to predict *a priori*, the amount of time required for garbage collection. CROPS5 does its own memory management and does not suffer from the problems associated with garbage collection.

CROPS5 also offers high performance from the perspective of parallelism. Its substrate, CParaOPS5, is a parallel version of OPS5 supporting match-level parallelism. CROPS5 extends this to support task level parallelism. Given a parallel processor, CROPS5 can potentially run in parallel.

Integration with Conventional Systems CROPS5 is very portable, and runs on most UNIX and Mach uniprocessor machines. Some parallel processors are also supported. CROPS5 also runs on the ARTS[TM89] and CHIMERA II[SSK90] real-time operating systems and is currently being ported to Real-Time Mach.

CROPS5 also supports mechanisms to facilitate easy integration between the rule-based component and with existing procedural software. A C-language interface is provided from the right-hand-sides of productions, allowing external C functions to access and modify internal Working Memory Elements of the production system. In addition, the environment interface allows CROPS5 to accept data directly from sensors, allowing CROPS5 to run on embedded platforms.

5 Evaluating the CROPS5 Architecture

This section looks at some of the performance measures of the CROPS5 architecture.

The time to detect an event depends on the granularity at which the system checks for external data. While previous systems recognized events on rule firing boundaries, our system can recognize events on token firing boundaries. The responsiveness of this system can be evaluated as the difference in the token processing time vs the typical rule firing time. Experiments with CROPS5 running on a SUN 3/60 under the CHIMERA II Real-Time Operating System provide the following average numbers for token firing and rule firing:

- Token Firing: 133.8 microseconds
- Rule Firing: 2350 microseconds

Based on the above numbers, we see that the time to recognize an event has improved by at least an order of magnitude.

Once an event has been recognized at a token boundary, the system can choose to ignore and continue processing the same stream, or it can elect to switch to a different stream. In our system, the times for these operations are given as:

- Ignore and continue processing: 9.41 microseconds
- Switch to a different stream: 54.83 microseconds

From these numbers we see that we pay a penalty of about 7.2 percent (9.41/133.8) if we pop up to check for new events at each token boundary. In return, we get an order of magnitude improvement in detection time.

If the input event is an alert-class event which is to be processed by a separate stream, our system can accomplish a stream switch in 54.83 microseconds. Since each stream has its own RETE net, processing can immediately start on the new stream. We do not incur the overhead of having to delete the current context element and matching the context element of the alert-class stream, as is done in current implementations. The deleting of the old context element, and the creation of the new context element takes a few match cycles. Based on the rule firing time of 2350 microseconds, the time to delete the old context element and creating a new one will take on the order of 4700 microseconds, depending on the number of rules in the affected contexts. Our stream switch time of 54.83 microseconds is about two orders of magnitude better.

The third element of responsiveness is the time to process the stream, and plan a response to the event. Since this element of responsiveness is very much situation and application dependent, we do not analyze it here. A paper in the upcoming August issue of the Communications of the ACM discusses these issues in greater depth.

CROPS5 has been used to develop test applications on the ARTS testbed. This includes a Collision Avoidance System for aircraft, and a prototype dynamic scheduling pacing system for steel mills.

6 Limitations

CROPS5 is a production system environment, and does not support full functionality for dynamically building rules at run-time. Hence, it has limited application in learning. Also, the language does not explicitly support representations and reasoning about time. This has to be done implicitly by the programmer. Current research is addressing these issues.

7 Current Work

We are currently working on extending this system, and are concentrating on developing a combination of architectural and problem-solving methodologies for integrating AI tasks into existing real-time systems. We are looking at scheduling approaches which will allow the AI task to run at any priority level in the system, without compromising the timing characteristics of the lower priority real-time tasks. We are currently developing models to characterize these facets of responsiveness. The eventual goal is to be able to *a priori* predict the responsiveness of the system to alert-class events. We are working on problem-solving and scheduling mechanisms to ensure both *inter-task* and *intra-task* stability.

As a result of this research, we hope to develop a combination of architectural features and problem-solving methodologies for building predictable problem-solving systems for real-time applications.

References

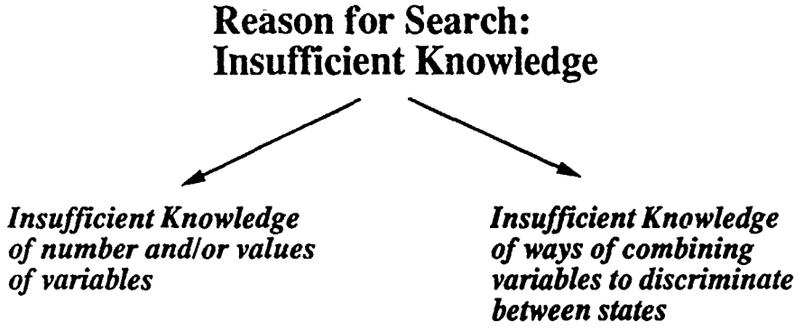
[BFKM85] Lee Brownston, Robert Farell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley Publishing Company, Inc., 1985.

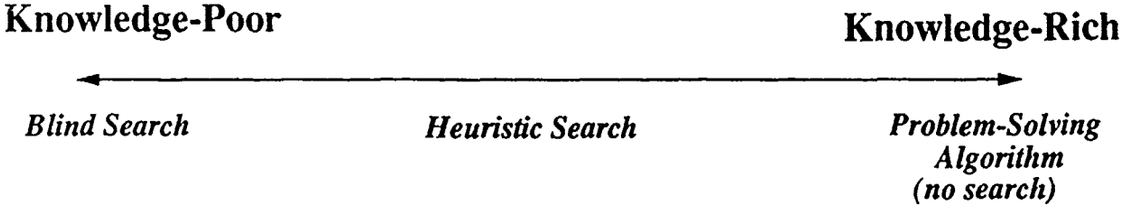
[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[KTG+88] D. Kalp, M. Tambe, A. Gupta, C. Forgy, A. Newell, A. Acharya, B. Milnes, and K. Swedlow. Parallel ops5 user's manual. Technical Report CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November 1988.

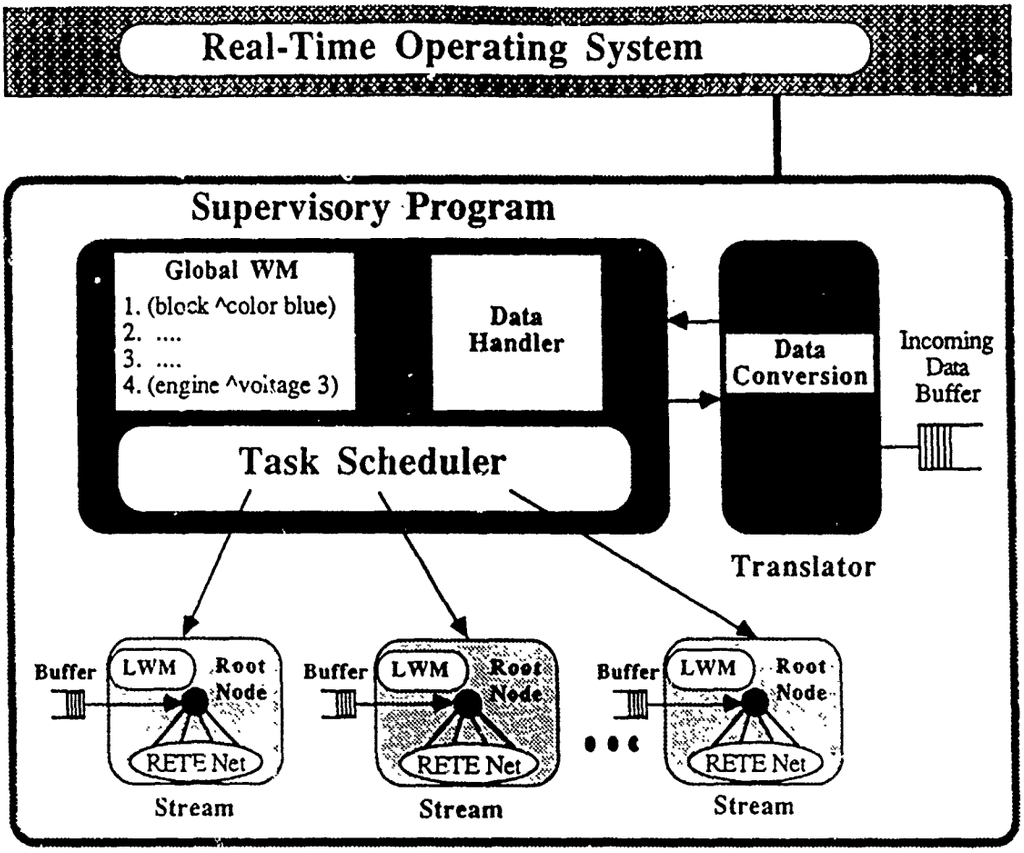
[SSK90] David Stewart, Donald E. Schmitz, and Pradeep Khosla. Implementing real-time robotic systems using chimera ii. In *Proceedings of the 1990 IEEE International Conference on Robotics And Automation*, pages 598–603, May 1990.

[TM89] Hideyuki Tokuda and Clifford Mercer. Arts: A distributed real-time kernel. *ACM Operating Systems Review*, 23(4), July 1989.





Search Spectrum


Real-Time Operating System
Supervisory Program
Global WM
1. (block ^color blue)
2. ....
3. ....
4. (engine ^voltage 3)
Data Handler
Task Scheduler
Data Conversion
Incoming Data Buffer
Translator
Buffer
LWM
Root Node
RETE Net
Stream
Buffer
LWM
Root Node
RETE Net
Stream
Buffer
LWM
Root Node
RETE Net
Stream

Limitations concerning on-line scheduling algorithms for overloaded real-time systems*

Sanjoy K. Baruah and Louis E. Rosier
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

With respect to on-line scheduling algorithms that must direct the service of sporadic task requests we quantify the benefit of possessing knowledge concerning the timing of future events. Consider the problem of preemptively scheduling sporadic task requests in a uniprocessor environment. If a task request is successfully scheduled to completion, a value equal to the request's execution time is obtained; otherwise a value of zero is obtained. We prove that no on-line scheduling algorithm can guarantee a cumulative value greater than $(\sqrt{2}-1)$ times the value obtainable by a clairvoyant scheduler, i.e., we prove a performance guarantee limitation of 41.4%. Over intervals where the loading factor does not exceed one the performance guarantee limitation is 100%. As the loading factor exceeds one we show that the performance guarantee limitation immediately drops to 61.8%, and as the loading factor increases from one to two, we show that the performance guarantee limitation falls from 61.8% to 41.4%. Beyond two the performance guarantee limitation remains at 41.4%.

1 Introduction.

Everyone no doubt believes that possessing knowledge concerning the timing of future events can be advantageous — but how advantageous? Within the domain of real-time scheduling we quantify the benefit of possessing knowledge about the future. Our treatment concerns on-line scheduling algorithms that must direct the service of sporadic task requests. An on-line scheduling algorithm is an iterative scheduling algorithm that makes a scheduling decision — about which task request to allocate a processor to —at each instant of time with no knowledge about what task requests will be made in the future. As a comparision vehicle, we consider clairvoyant on-line scheduling algorithms that know the arrival time, computation time and deadline of all future task requests. We then quantify the value of knowing the future by establishing limitations concerning how responsive on-line algorithms can be as compared to their clairvoyant counterparts.

We consider the problem of preemptively scheduling sporadic task requests in a uniprocessor environment. If a task request is successfully scheduled to completion, a value equal to the request's execution time is obtained; otherwise a value of zero is obtained. A sporadic real-time environment is said to have a **loading factor** b iff it is guaranteed that there will be no interval of time $[t_x, t_y)$ such that the sums of the execution-times of all task-requests making requests and having deadlines within this interval is greater than $b \cdot (t_y - t_x)$. A sporadic real-time environment is **overloaded** unless it has a loading factor no greater than 1. Liu and Layland's deadline algorithm [3] constitutes an optimal *on-line* scheduling algorithm for *feasible* task systems [1], and for all task systems during intervals where the system is not overloaded [2]. (A scheduling algorithm is *optimal* if it generates the maximum possible cumulative value — the value obtainable by the cleverest clairvoyant algorithm.) A number of other algorithms work well — though not optimally — during non-overloaded intervals. These range from the ever-popular rate-monotonic algorithm [3] to the various scheduling schemes tested by Locke [4].

Experimental results suggest that the algorithms mentioned above perform extremely poorly during overloaded intervals, however [4]. As a result, Locke [4] designed a dynamic priority scheduling algorithm that experimentally

*This work was supported in part by U.S. Office of Naval Research Grant No. N00014-89-J-1913.

performs quite well even though there are pathological situations where it performs very badly. The algorithm, known as Locke's Best-effort scheduler, has one major drawback — it meets no absolute performance guarantee. That is, it is not guaranteed to behave in a manner that will generate a cumulative value that comes within a constant factor of the maximum possible. Recently, Mishra, Raghunathan, and Shasha [5] unveiled an on-line scheduling algorithm that always achieves optimal performance during non-overloaded intervals and 25% of the optimal performance during overloaded intervals. Furthermore, the algorithm experimentally compares well with Locke's Best-effort scheduler.

A performance guarantee of 25% may not seem extremely good, but we will show that one cannot do much better. In Section 2, we prove that no on-line scheduling algorithm can have a performance guarantee greater than $(\sqrt{2}-1)$ times the optimal — about 41.4%. We also generalise this result by considering environments where there is an upper bound on the amount of overloading allowed within an interval, i.e., a bound on the loading factor within an interval. Whenever the loading factor does not exceed one the performance guarantee limitation is 100% — obviously. As the loading factor exceeds one we show that the performance guarantee limitation immediately drops to 61.8%, and as the loading factor increases from one to two, we show that the performance guarantee limitation falls from 61.8% to 41.4%. Beyond two the performance guarantee limitation remains at 41.4%. We conclude in Section 3 with a discussion of future research.

2 Performance Guarantee Limitations

Consider a uniprocessor environment as defined by Locke [4] and Mishra Raghunathan, & Shasha [5], where task requests are characterised by two parameters — an *execution time* e and a *deadline* d. If task request $T = (e, d)$ makes a request for the processor at time t_0, then a value equal to the execution time e is obtained by allocating the processor to T for e units of time in the interval $\{i | t_0 \leq t < t_0 + d\}$. Failure to allocate the processor to T for e units of time in this interval results in a value of zero. An **on-line scheduling algorithm** is an iterative algorithm that makes a scheduling decision — i.e., decides which request to allocate the processor to — at each time instant with no knowledge about what requests will be made in the future. An on-line algorithm has **performance guarantee** r, $0 \leq r \leq 1$, iff it is guaranteed to achieve a cumulative value at least r times the cumulative value achievable by a clairvoyant algorithm on *any* sequence of task requests. (Hence the Mishra, Raghunathan, and Shasha [5] algorithm has a performance guarantee of 25%, and Locke's Best-effort scheduler has a performance guarantee of 0%.)

Lemma 1 *There does not exist an on-line scheduling algorithm with a performance guarantee greater than* $(\sqrt{2}-1)$ *— about* 41.4%.

Proof : Our proof is via an adversarial argument. We illustrate a situation such that, no matter what decision a scheduling algorithm makes, there is always a sequence of task requests for which the ratio of the performance of this algorithm to the performance of a clairvoyant scheduling algorithm is bounded from above by $\sqrt{2}-1$.

Consider the following scenario:
Let t be the current time. At time $(t - e_1 + \delta_1)$ (where δ_1 is a very small positive real number; $\delta_1 \ll e_1$), task-request $T_1 = (e_1, e_1)$ made a request and was allocated the processor. T_1 will therefore complete execution at time $(t + \delta_1)$ if allowed to execute without interruption. Currently there are no other requests in the system. At time t, a task $T_2 = (e_2, e_2)$, $e_2 \geq e_1$, makes a request. Clearly, no scheduling algorithm can hope to successfully schedule both requests T_1 and T_2. We consider the following two cases:

Case A *The on-line scheduling algorithm continues the execution of request* T_1.
In the event of there being no other requests ever made to the system, the performance of this algorithm is e_1/e_2 times that of a clairvoyant scheduler. The performance guarantee of this algorithm is therefore no greater than e_1/e_2.

Case B *The on-line scheduling algorithm preempts* T_1, *and commences the execution of* T_2.
Consider an adversary who makes a sequence of requests based on the decisions made by the on-line algorithm, as described below:

```
begin
/* Variable clock represents the current time */
/* Let e3 be a very small positive real number; e3 ≪ e1 and e2.  */
wait until clock = t + δ1;
while clock + e3 < (t + e2) loop
        Task-request T3 = (e3, e3) makes a request;
        if the on-line algorithm preempts task T2 in favour of task T3, then goto finis;
        wait e3 units:
        end loop
Task request T4 = (e2, e2) makes a request;
finis : no further requests are made;
end.
```

If the loop is exited by means of the goto statement, then the performance of the on-line algorithm is $\leq e_3/e_2$ times that of a clairvoyant scheduler on this particular request sequence. If the loop is exited the normal way (T_4 makes a request in the time interval $\{t' : (t + e_2 - e_3) < t' < (t + e_2)\}$), then, for this sequence of requests, a clairvoyant scheduling algorithm would choose to execute T_1, all the requests T_3 and the request T_4. Thus, the clairvoyant algorithm gets a value $e_1 + e_2+$ (the sums of the values of the requests T_3). By an appropriate choice of δ_1 and e_3, the third term in this sum may be made arbitrarily close to e_2. In contrast, the on-line algorithm may schedule either T_2 or T_4 — for a net value of e_2. Its performance is therefore no more than $e_2/(e_1 + 2e_2)$ times that of the clairvoyant scheduling algorithm. For $\frac{e_1}{e_2} = (\sqrt{2} - 1)$, therefore, the performance guarantee for both case A and case B are $\leq (\sqrt{2} - 1)$. For $\frac{e_1}{e_2} = (\sqrt{2} - 1)$, therefore, no on-line algorithm can make a performance guarantee greater than $(\sqrt{2} - 1)$. This proves the lemma.

□

In deriving the upper bound above, an adversary argument was used wherein the malevolent adversary was allowed to introduce new tasks at will. For example, there were requests for a total of $2e_2$ units of processor time with both request-times and deadlines in the time-interval $[t, t+e_2)$ — an interval of size less than e_2. A natural question to ask at this stage would be — is the *amount* of overloading permitted by the environment related to the best performance guarantee that may be guaranteed by an on-line algorithm? To answer this question, let us quantify the notion of overloading.

We say a sporadic real-time environment has a **loading factor** b iff it is guaranteed that there will be no interval of time $[t_x, t_y)$ such that the sums of the execution-times of all task-requests making requests and having deadlines within this interval is greater than $b \cdot (t_y - t_x)$. A sporadic real-time environment is **overloaded** unless it has a loading factor no greater than 1.

Example 1 : The sporadic real-time environment discussed in the proof of Lemma 1 must have a loading factor greater than 2. This is since the sums of the execution-times of all requests that the adversary comes up with in Case B, along with the request T_2, with both request-times and deadlines in the time-interval $[t, t + e_2)$ is $2e_2$.

□

Notice that the loading factor is defined with respect to an environment and not with respect to a sequence of task requests — this is important. The on-line algorithm knows *a priori* what the loading factor for the environment is, and may use this information in making on-line scheduling decisions. Consider, as an example, on-line scheduling in an environment which is known to have a loading factor no larger than 1 (i.e., a non-overloaded environment). Dertouzos [2] has shown that the Deadline algorithm is optimal in such an environment. The deadline algorithm is, therefore, an on-line scheduler with a performance guarantee of 1.0 in sporadic real-time environments with a loading factor no larger than 1. At the other extreme, Lemma 1 proves that no on-line scheduler can offer a performance guarantee larger than .414 in environments where the loading factor may be larger than 2 (see Example 1).

The following lemma quantifies the relationship between the loading factor and the upper bound on the performance guarantee of an on-line algorithm in environments where the loading factor is between 1 and 2.

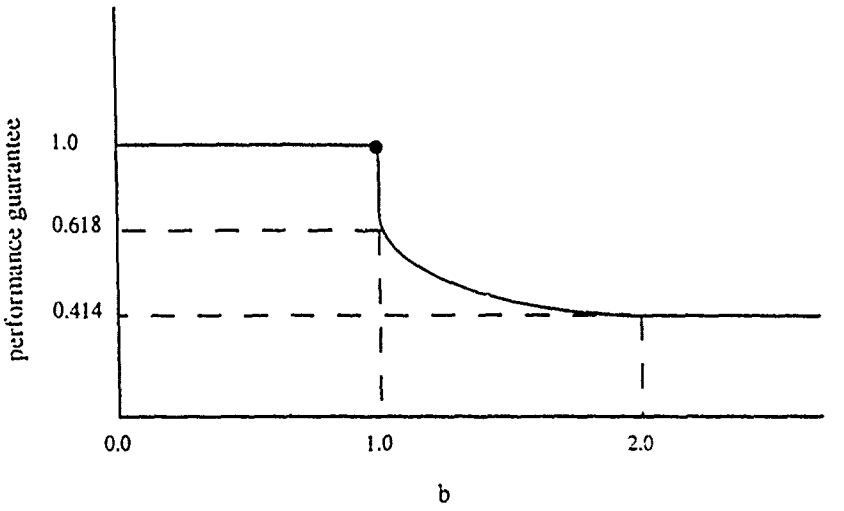


Figure 1: Performance guarantee as a function of the loading factor

Lemma 2 *No on-line scheduling algorithm operating in an environment with a loading factor* b, $1 < b \leq 2$, *can make a performance guarantee greater than*

$$\frac{\sqrt{b^2+4}-b}{2}.$$

Proof : The proof of this lemma mirrors the one for Lemma 1. Consider the same scenario as in Lemma 1, with the following change

- Task-request $T_3 = ((b-1)\cdot e_3, e_3)$ (as before, e_3 is a very small positive real number: $e_3 \ll e_1$ and e_2).

In this (modified) scenario, the sums of the execution-times of all requests with both request-times and deadlines in the time-interval $[t, t+e_2)$ in Case B is $e_2+(b-1)\cdot e_2 = b\cdot e_2$. The adversary's request-sequence does not, therefore, violate the loading-factor of the environment.

When $\frac{e_1}{e_2} = (\sqrt{b^2+4}-b)/2$, the performance guarantee for both case A and case B are $\leq (\sqrt{b^2+4}-b)/2$. For $\frac{e_1}{e_2} = (\sqrt{b^2+4}-b)/2$; therefore, no on-line algorithm can make a performance guarantee greater than $(\sqrt{b^2+4}-b)/2$. This proves the lemma.

□

The Deadline algorithm is an on-line scheduling algorithm with a performance guarantee of 1 for environments where the loading factor b is at most 1.0. For environments with $1 < b \leq 2$, Lemma 2 provides an upper bound for the performance guarantee of on-line algorithms, while 0.414 is an upper bound for the performance guarantee of on-line algorithms in environments where $b \geq 2$ (see Lemma 1). These results are summarised in the following theorem:

Theorem 1 *For a loading factor* $b \leq 1$, *there exist on-line schedulers which make a performance guarantee of* 1.0. *For* $b = 1+\epsilon$ *where* ϵ *is an arbitrary small positive number,* 0.618 *is an upper bound on the performance guarantee of any on-line scheduler. For* $1 < b \leq 2$, $(\sqrt{b^2+4}-b)/2$ *is an upper bound on the performance guarantee of any on-line scheduler, and for* $b > 2$, *no on-line scheduler can make a performance guarantee greater than* 0.414.

Figure 1 plots the upper bounds on the performance guarantee described in Theorem 1 as a function of the loading factor b.

3 Conclusions and directions for future research

We have studied the performance limitations of on-line algorithms in environments where a value equal to the request's execution time is obtained for the full execution of a request and no value is obtained for the partial execution of a request. One might argue that it may be more realistic to consider systems where some task-requests are more important than others, i.e., the value obtained by completing the execution of a task-request depends on the importance of the request to the system as well as the execution-time of the request. To model systems of this type, each request is parametrised by three parameters: the *importance* i in addition to the execution-time e and the deadline d. By successfully executing task-request $T = (i, e, d)$, a value $i \times e$ is obtained (as before, no value is obtained for incomplete execution of a task). All of our results can be generalised for such systems. Unfortunately, the performance guarantees that can be made by an on-line algorithm in such an environment is even less than what's stated above. We will elaborate on this in an expanded version of this paper currently under preparation.

Recently G. Koren, B. Mishra, A. Raghunathan, and D. Shasha have revised [5], yielding a slightly reduced performance guarantee. More recently, we in collaboration with Koren, Mishra, Raghunathan, and Shasha have made substantial improvements in the upper bounds reported herein. A manuscript discussing all of these new results will soon be available.

References

[1] S Baruah, A. Mok, and L. Rosier. The preemptive scheduling of sporadic, real-time tasks on one processor. In *Eleventh Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990.

[2] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[3] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[4] C. D. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[5] B. Mishra, A. Raghunathan, and D. Shasha. A competitive on-line algorithm for overloaded real-time systems (extended abstract). Unpublished manuscript, 1990.

HARD REAL-TIME SCHEDULING: THE DEADLINE-MONOTONIC APPROACH[1]

N.C. Audsley A. Burns M. F. Richardson A.J. Wellings

Department of Computer Science,
University of York,
York.
YO1 5DD
England.
neil@uk.ac.york.minster

1. INTRODUCTION

A real-time system is one in which failure can occur in the time domain as well as in the more familiar value domain. If the consequence of such failure is catastrophic then the system is often referred to as a *hard real-time system.* Such systems are needed in a number of application domains including air-traffic control, process control, and numerous embedded systems.

In the development of application programs it is usual to map system timing requirements onto process deadlines. The issue of meeting deadlines therefore becomes one of process scheduling. The development of appropriate scheduling algorithms has been isolated as one of the crucial challenges for the next generation of real-time systems [9].

One scheduling method that is used in hard real-time systems is based upon rate-monotonic theory [6]. At runtime a pre-emptive scheduling mechanism is used: the highest priority runnable process is executed. Priorities assigned to processes are inversley proportional to the length of period. That is, the process with the shortest period is assigned the highest priority. Rate-monotonic scheduling has several useful properties, including a simple "sufficient and not necessary" schedulabilty test based on process utilisations [6]; and a complex sufficient and necessary schedulability test [4]. However, the constraints that it imposes on the process set are severe: processes must be periodic, independent and have deadline equal to period.

Many papers have successively weakened the constraints imposed by the rate-monotonic approach and have provided associated schedulability tests. Reported work includes a test to allow aperiodic processes to be included in the theory [8], and a test to incorporate processes that synchronise using semaphores [7]. One constraint that has remained within rate-monotonic literature is that the deadline and period of a process must be equal.

Deadline-monotonic [5] priority assignment weakens this constraint within a static priority scheduling scheme. However, no schedulability tests were given in[5] for the scheme.

The weakening of the "period equals deadline" constraint would benefit the application designer by providing a more flexible process model. For example, precedence constraints in a distributed system can be modelled as a sequence of periodic processes (one per processor). The inevitable communication delay is modelled as an interval of "dead time" at the end of each processes period (apart from the last process in the sequence). These periodic processes must therefore complete their computations by a deadline that is before the end of the period. Such a model has been used to good effect in a process allocation scheme [10] in which network communication overhead is traded against local schedulability. The greater the inter-processor traffic the greater the "dead time" and hence the lower the schedulability bound. Tindell's analysis [10] uses the schedulability tests discussed in this paper.

Another important motivation for weakening the "deadline equal to period" constraint is to cater for sporadic (aperiodic) events in an efficient manner. Here the required response time is not, in general, related to the worst case arrival rate. Indeed the characteristics of such events often demand a short response time compared to minimum inter-arrival time. Hence polling in a periodic manner for sporadic events produces a non-optimal respoose time for sporadic processes.

This paper outlines the deadline-montonic scheduling approach together with new simple and complex schedulability tests that are sufficient and in the latter case, necessary. The approach is then shown to encompass sporadic processes that have hard deadlines without any alteration to the theory and without resorting to the inefficiencies of a polling approach.

2. DEADLINE-MONOTONIC SCHEDULING THEORY

We begin by observing that the processes we wish to schedule are characterised by the following relationship:

$$computation\ time \leq deadline \leq period$$

i.e. for each process i (where process 1 has the highest priority and process n the lowest in a system containing n processes):

$$C_i \leq D_i \leq T_i$$

where C gives computation time, D the deadline and T the period of process i.

Leung *et al* [5] have defined a priority assignment scheme that caters for processes with the above relationship. This is termed inverse-

1. This work is supported, in part, by the Information Engineering Advanced Technology Programme, Grant GR/F 35920/4/1/1214

deadline or deadline-monotonic priority assignment. No schedulability tests were given however.

Deadline-monotonic priority ordering is similar in concept to rate-monotonic priority ordering. Priorities assigned to processes are inversely proportional to the length of the deadline [5]. Thus, the process with the shortest deadline is assigned the highest priority and the longest deadline process is assigned the lowest priority. This priority ordering defaults to a rate-monotonic ordering when *period=deadline*.

Deadline-monotonic priority assignment is an optimal static priority scheme (see theorem 2.4 in [5]). The implication of this is that if any static priority scheduling algorithm can schedule a process set where process deadlines are unequal to their periods, an algorithm using deadline-monotonic priority ordering for processes will also schedule that process set.

It is true, of course, that any process sets whose timing characteristics are suitable for rate-monotonic analysis would also be accepted by a static priority theory permitting deadlines and periods of a process to differ.

In general the deadline-monotonic scheme has not been employed because of the lack of adequate schedulability tests. Rate-monotonic scheduling schedulability tests could be used by reducing the period of individual processes until equal to the deadline. Obviously such tests would not be optimal as the workload on the processor would be over-estimated.

New schedulability tests have been developed by the authors for the deadline-monotonic approach [1]. These tests are founded upon the concept of *critical instants* [6]. These represent the times that all processes are released simultaneously. When such an event occurs, we have the worst-case processor demand. Implicitly, if all processes can meet their deadlines for executions beginning at a critical instant, then they will always meet their deadlines. Thus, we have formed the basis for a schedulabilty test: check the executions of all processes for a single execution assuming that all processes are released simultaneously. One such schedulability test is given by[2]:

$$\forall i : 1 \le i \le n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \le 1 \qquad (1)$$

where I_i is a measure of higher priority processes interfering with the execution of τ_i:[3]

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

The test states that for a process τ_i to be schedulable, the sum of its computation time and the interference that is imposed upon it by higher priority processes executing must be no more than D_i. In the above, the interference is composed of the computation time of all higher priority processes that are released before the deadline of τ_i. This test is sufficient, but not necessary for the following reason. When the interference is being calculated, account is taken of executions of higher priority processes that start before D_i and could possibly complete execution after D_i. Therefore, I_i could be greater than the actual interference encountered by τ_i before D_i.

A more accurate test is given by[4]:

$$\forall i : 1 \le i \le n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \le 1 \qquad (2)$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \left\lceil \frac{D_i}{T_j} \right\rceil C_j + \min \left(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right) \right\rceil$$

The above test compensates within I_i for the parts of executions of higher priority processes that could not occur before D_i even though they were released before D_i. The test is not necessary as a pessimistic valuation is made of the time that will be utilised by higher priority processes before D_i.

The schedulability constraints given by equations (1) and (2) are sufficient but not necessary in the general case. To form a sufficient and necessary schedulability test the schedule has to be evaluated so that the exact interleaving of higher priority process executions is known. This is costly as this would require the solution of D_i equations per process τ_i.

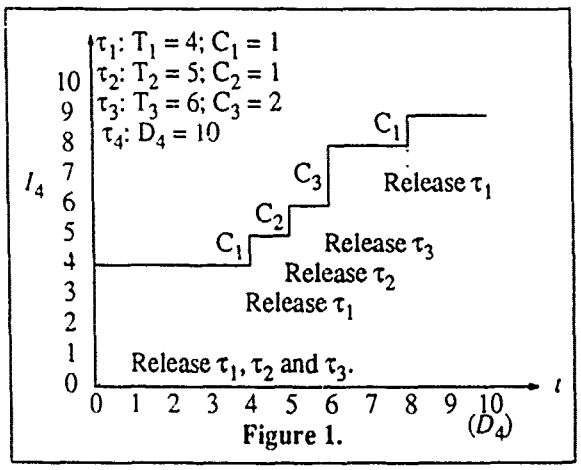


Figure 1.

The number of equations can however be reduced by observing that if τ_i meets its deadline at t'_i, where t'_i lies in $[0, D_i]$, we need not evaluate the equations in $(t'_i, D_i]$. Further reductions in the number of equations requiring solution can be made by limiting the points in $[0, D_i]$ that are considered as possible solutions for t'_i. Consider the times within $[0, D_i]$ that τ_i could possibly meet its deadline. We note that the interference due to high-priority processes is monotonically increasing within this interval. The points in time that the interference increases occur when there is a release of a higher priority process. This is illustrated by Figure 1. In the figure there are three processes with higher priority than τ_4. We see that as the higher priority processes are

2,4. for derivation see [1].

3. Note: $\lceil x \rceil$ evaluates to the smallest integer $\ge x$; $\lfloor x \rfloor$ evaluates to the largest integer $\le x$

released, I_4 increases monotonically with respect to t. The graph is stepped with plateaus representing intervals of time in which no higher priority processes are released. It is obvious that only one equation need be evaluated for each plateau as the interference does not change.

To maximise the time available for the execution of τ_i we choose to evaluate at the rightmost point on the plateau. Therefore, one possible reduction in the number of equations to evaluate schedulability occurs by testing τ_i at all points in $[0, D_i]$ that correspond to a higher priority process release. Since as soon as one equation identifies the process set as schedulable we need test no further equations. Thus, the effect is to evaluate equations only in $[0, t'_i]$.

The number of equations has been reduced in most cases. We note that no reduction will occur if for each point in time in $[0, D_i)$ a higher priority process is released with τ_i meeting its deadline at D_i. The number of equations is reduced yet further by considering the computation times of the processes. Consider Figure 2.

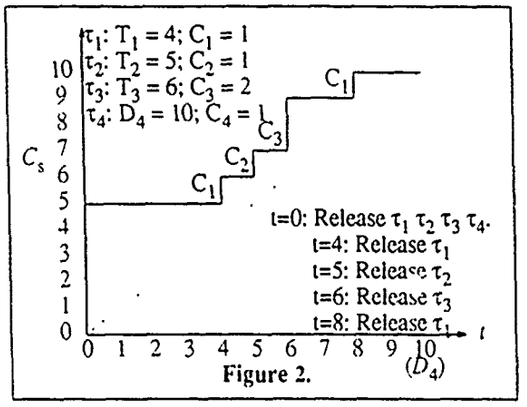


Figure 2.

In Figure 2 the total computation requirement of the process set (C_s) is plotted against time. At the first point in time when the outstanding computation is equal to the time elapsed, we have found t'_4. In the above diagram this point in time coincides with the deadline of τ_4.

Considering Figure 2, there is little merit in testing the schedulability of τ_i in the interval $[0, C_i)$. Also, since time 0 corresponds with a critical instant (a simultaneous release of all processes) the first point in time that τ_i could possibly complete is:

$$t_0 = \sum_{j=1}^{i} C_i$$

This gives a schedulability constraint of:

$$\frac{I_i^{t_0}}{t_0} + \frac{C_i}{t_0} \leq 1$$

where

$$I_y^x = \sum_{z=1}^{y-1} \left\lceil \frac{x}{T_z} \right\rceil C_z$$

Since the value of t_1 assumes that only one release of each process occurs in $\left[0, t_0\right)$, the constraint will fail if there have been any releases of higher priority processes within the interval $\left[0, t_0\right]$. The exact amount of work created by higher priority processes in this interval is given by:

$$I_i^{t_0}$$

The next point in time at which τ_i may complete execution is:

$$t_1 = I_i^{t_0} + C_i$$

This gives a schedulability constraint of:

$$\frac{I_i^{t_1}}{t_1} + \frac{C_i}{t_1} \leq 1$$

Again, the constraint will fail if releases have occurred in the interval $\left[t_0, t_1\right)$. Thus, we can build a series of equations to express the schedulability of τ_i.

$$(1) \quad \frac{I_i^{t_0}}{t_0} + \frac{C_i}{t_0} \leq 1$$

$$\text{where} \quad t_0 = \sum_{j=1}^{i} C_j$$

$$(2) \quad \frac{I_i^{t_1}}{t_1} + \frac{C_i}{t_1} \leq 1$$

$$\text{where} \quad t_1 = I_i^{t_0} + C_i$$

$$(3) \quad \frac{I_i^{t_2}}{t_2} + \frac{C_i}{t_2} \leq 1$$

$$\text{where} \quad t_2 = I_i^{t_1} + C_i$$

$$\vdots$$

$$(k) \quad \frac{I_i^{t_k}}{t_k} + \frac{C_i}{t_k} \leq 1$$

$$\text{where} \quad t_k = I_i^{t_{k-1}} + C_i$$

and where

$$I_y^x = \sum_{z=1}^{y-1} \left\lceil \frac{x}{T_z} \right\rceil C_z$$

If any of the equations hold, τ_i is schedulable. Obviously, the equations terminate if $t_k > D_i$ for process τ_i and equation k. At this point τ_i is unschedulable.

The series of equations above is encapsulated by the algorithm given in Figure 3. The algorithm progresses since the following relation always holds:

$$t_i > t_{i-1}$$

When t_i is greater than D_i the algorithm terminates since τ_i is unschedulable. Thus we have a maximum number of steps of D_i. This is a worst-case measure. We note that the algorithm can be used to evaulate the schedulability of any fixed priority process set where process deadlines are no greater than periods, whatever the assignment rule used for priorities.

Algorithm

foreach τ_i *do*

$t = \sum_{j=1}^{i} C_j$

continue = *TRUE*

while [*continue*] *do*

if $\left[\frac{I_i^t}{t} + \frac{C_i}{t} \leq 1 \right]$

continue = *FALSE*

/* *NB* τ_i *is schedulable* */

else

$t = I_i^t + C_i$

endif

if [$t > D_i$]

exit

/* *NB* τ_i *is unschedulable* */

endif

endwhile

endfor

■

Figure 3.

3. SCHEDULING SPORADIC PROCESSES

Non-periodic processes are those whose releases are not periodic in nature. Such processes can be subdivided into two categories [2]: aperiodic and sporadic. The difference between these categories lies in the nature of their release frequencies. Aperiodic processes are those whose release frequency is unbounded. In the extreme, this could lead to an arbitrarily large number of simultaneously active processes. Sporadic processes are those that have a maximum frequency such that only one instance of a particular sporadic process can be active at a time.

When a static scheduling algorithm is employed, it is difficult to introduce non-periodic process executions into the schedule: it is not known before the system is run when non-periodic processes will be released. More difficulties arise when attempting to guarantee the deadlines of those processes. It is clearly impossible to guarantee the deadlines of aperiodic processes as there could be an arbitrarily large number of them active at any time. Sporadic processes deadlines can be guaranteed since it is possible, by means of the maximum release frequency, to define the maximum workload they place upon the system.

One approach is to use static periodic polling processes to provide sporadics with executions time. This approach is reviewed in section 3.1. Section 3.2 illustrates how to utilise the properties of the deadline monotonic scheduling algorithm to guarantee the deadlines of sporadic processes without resorting to the introduction of polling processes.

3.1. Sporadic Processes: the Polling Approach

To allow sporadic processes to execute within the confines of a static schedule (such as that generated by the rate-monotonic algorithm) computation time must be reserved within that schedule. An intuitive solution is to set up a periodic process which polls for sporadic processes [3]. Strict polling reduces the bandwidth of processing as

- processing time that is embodied in an execution of the polling process is wasted if no sporadic process is active when the polling process becomes runnable;
- sporadic processes occurring after the polling process's computation time in one period has been exhausted or just passed have to wait until the next period for service.

A number of bandwidth preserving algorithms have been proposed for use with the rate-monotonic scheduling algorithm [3, 8]. These algorithms are founded upon a periodic server process being allotted a number of units of computation time per period. These units can be used by any sporadic process with outstanding computational requirements. The computation time for the server is replenished at the start of its period.

Problems arise when sporadic processes require deadlines to be guaranteed. It is difficult to accommodate these within periodic server processes due to the rigidly defined points in time at which the server computation time is replenished. The sporadic server [8] provides a solution to this problem. The replenishment times are related to when the sporadic uses computation time rather than merely at the period of the server process. However, this approach still requires additional processes with obvious extra overheads.

3.2. Sporadic Processes: the Deadline Monotonic Scheduling Approach

We now show how deadlines of sporadic processes can be guaranteed within the existing deadline-monotonic theory. Consider the timing characteristics of a sporadic process τ_s. The demand for computation time is illustrated in Figure 4.

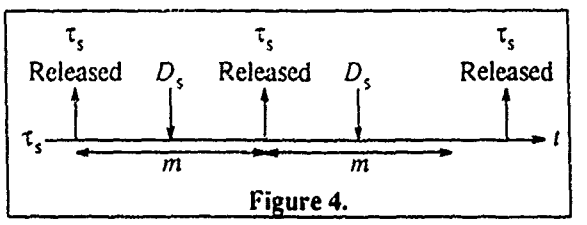


Figure 4.

The minimum time difference between successive releases of τ_s is the minimum inter-arrival time m. This occurs between the first two releases

of τ_s. At this point, τ_s is behaving exactly like a periodic process with period m: the sporadic is being released at its maximum frequency and so is imposing its maximum workload. When the releases do not occur at the maximum rate (between the second and third releases in Figure 4) τ_s behaves like a periodic process that is intermittently activated and then laid dormant. The workload imposed by the sporadic is at a maximum when the process is released, but falls when the next release occurs after greater than m time units have elapsed.

In the worst-case the τ_s behaves exactly like a periodic process with period m and deadline D where $D \leq m$. The characteristic of this behaviour is that a maximum of one release of the process can occur in any interval $[t, t+m]$ where release time t is at least m time units after the previous release of the process. This implies that to guarantee the deadline of the sporadic process the computation time must be available within the interval $[t, t+D]$ noting that the deadline will be at least m after the previous deadline of the sporadic. This is exactly the guarantee given by the deadline-monotonic schedulability tests in section 2.

For schedulability purposes only, we can describe the sporadic process as a periodic process whose period is equal to m. However, we note that since the process is sporadic, the actual release times of the process will not be periodic, but successive releases will be separated by no less than m time units.

For the schedulability tests given in section 2 to be effective for this process system, we assume that at some instant all processes, both periodic and sporadic, are released simultaneously (i.e. a critical instant). We assume that this occurs at time 0. If the deadline of the sporadic can be guaranteed for the release at a critical instant then all subsequent deadlines are guaranteed. Examples of this approach are g ,n in [1]. No limitations on the combination of periodic and sporadic processes are imposed by this scheme. Indeed, the approach is optimal for a fixed priority scheduling since sporadic processes are treated in exactly the same manner as periodic processes. All three schedulability tests outlined in section 2 are suitable for use with sporadic processes. To improve the responsiveness of sporadic proceses their deadlines can be reduced to the point at which the system becomes unschedulable.

4. CONCLUSIONS

The fundamental constraint of rate-monotonic scheduling theory has been weakened to permit processes that have deadlines less than period to be scheduled. The result is the deadline-monotonic scheduling theory. Schedulability tests have been presented for the theory.

Initially a simple sufficient and not necessary schedulability test was introduced. This required a single equation per process to determine schedulability. However, to achieve such simplicity meant the test was overly pessimistic. The simplifications made to produce a single equation test were then partially removed. This produced a sufficient and not necessary schedulability test which passed more process sets than the simple test. Again, the test was pessimistic.

This problem was resolved with the development of a sufficient and necessary schedulability test. This was the most complex of all the tests having a complexity related to the periods and computation times of the processes in the set. The complexity was reduced substantially when the number of equations required to determine the schedulability of a process were minimised. This test is able to determine the schedulability of any fixed priority process set where deadlines are no greater than periods, whatever the priority assignment criteria used.

Proposed methods for guaranteeing deadlines of sporadic processes using sporadic servers within the rate-monotonic scheduling framework were shown to have two main drawbacks. Firstly, one extra periodic server process is required for each sporadic process. Secondly, an extra run-time overhead is created as the kernel is required to keep track of the exact amount of time the server has left within any period. The deadline-monotonic approach circumvents these problems since no extra processes are required: the sporadic processes can be dealt with adequately within the existing periodic framework.

A number of issues raised by the work outlined in this paper require further consideration. These include the effect of allowing processes to synchronise and vary their timing characteristics. These issues remain for further investigation, although it is the authors' contention that the analysis that has been focussed upon the rate-monotonic approach shows that deadline-monotonic schedulability theory is easily extensible to address such issues.

REFERENCES

1. N. C. Audsley, "Deadline Monotonic Scheduling", YCS 146, Dept. of Comp. Sci., Univ. of York (1990).
2. A. Burns, "Scheduling Hard Real-Time Systems: A Review", *Software Eng. Journal (to appear)* (1991).
3. J. P. Lehoczky, L. Sha and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *Proc IEEE Real-Time Sys. Symp.*, pp. 261-270 (1987).
4. J. Lehoczky, L. Sha and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour", *Proc IEEE Real-Time Sys. Symp.*, pp. 166-171 (1989).
5. J. Y. T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", *Perf. Eval. (Netherlands)* 2(4), pp. 237-250 (1982).
6. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *JACM* 20(1), pp. 40-61 (1973).
7. L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada", CMU/SEI-88-TR-33, SEI, Carnegie-Mellon University (1988).
8. L. Sha, B. Sprunt and J. P. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems", *Journal of Real-Time Sys.* 1, pp. 27-69 (1989).
9. J.A. Stankovic, "Real-Time Computing Systems: The Next Generation", COINS Tech. Rep. 88-06, Dept. of Comp. and Inf. Sci., Univ. of Massachusetts (1988).
10. K. Tindell, "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)", YCS 147, Dept. of Comp. Sci., Univ. of York (1990).

Algorithms for Flow-Shop Scheduling to Meet Deadlines

R. Bettati and Jane W.S. Liu

Department of Computer Science
University of Illinois, Urbana, IL 61801, USA

1. Introduction

A *flow shop* [1-6] models a multiprocessor or distributed system in which processors and devices (also modeled as processors) are functionally dedicated. Each task executes on the processors in turn, following the same order. For example, a real-time control system containing an input processor, a computation processor, and an output processor can be modeled as a flow shop if each task in the system executes first on the input processor, then on the computation processor, and finally on the output processor. Many hard real-time systems can be modeled as flow shops in which every task has a *deadline*, the point in time by which its execution must be completed. The primary objective of scheduling in such systems is to find schedules in which all tasks meet their deadlines whenever such schedules, called *feasible schedules*, exist. A scheduling algorithm is said to be *optimal* if it always finds a feasible schedule whenever feasible schedules exist.

The problem of scheduling tasks in flow shops to meet deadlines is NP-hard, except for a few special cases [1,2]. We describe here a heuristic algorithm for scheduling tasks with arbitrary processing times and discuss its performance. We also consider two variations of the traditional flow-shop model, called *flow shop with recurrence* and *periodic flow shop*. In a flow shop with recurrence, each task executes more than once on one or more processors. A system that does not have a dedicated processor for every function can often be modeled as a flow shop with recurrence. As an example, suppose that the three processors in the control system mentioned earlier are connected by a bus. We can model the bus as a processor and the system as a flow shop with recurrence. Each task executes first on the input processor, then on the bus, on the computation processor, on the bus again, and finally on the output processor. In a periodic flow shop, each task is a periodic sequence of requests for the same computation.

In the following, we begin by describing the flow-shop models in Section 2. Section 3 describes a heuristic algorithm for scheduling tasks with arbitrary processing times and the simulation results on its performance. Section 4 describes an optimal algorithm for scheduling in flow shops with recurrence where tasks have identical processing times and ready times. Section 5 describes a way for scheduling in periodic flow shops. Section 6 is a summary.

2. Flow Shop Models

In (traditional) flow shops, there are m different processors $P_1, P_2, \cdots, P_m$. We are given a task set $\mathbf{T}$ of n tasks $T_1, T_2, \cdots, T_n$. Each task T_i consists of m subtasks $T_{i1}, T_{i2}, \cdots, T_{im}$. These subtasks have to be executed in order: the subtask T_{ij} executes on processor P_j after the subtask $T_{i(j-1)}$ completes on processor $P_{(j-1)}$, for $j = 2, 3, \cdots, m$. Task T_i is ready for execution at or after its *release time* r_i and must be completed by its *deadline* d_i. Let τ_{ij} denote the *processing time* of T_{ij}, the time required for the subtask T_{ij} to complete its execution. We occasionally refer to the totality of release times, deadlines, and processing times as the *task parameters*. The task parameters are rational numbers unless it is stated otherwise.

In the more general flow-shop-with-recurrence model, each task T_i in $\mathbf{T}$ has k subtasks, and $k > m$. The subtasks are executed in the order $T_{i1}, T_{i2}, \cdots, T_{ik}$. We characterize the order in which the subtasks execute on the processors by a sequence $V = (v_1, v_2, \cdots, v_k)$ of integers, where v_j is one of the integers in the set $\{1, 2, \cdots, m\}$. v_j being l means that the subtasks T_{ij} are executed on processor P_l. For example, suppose that each task in the given set $\mathbf{T}$ has 5 subtasks to be executed on 4 processors. The sequence $V =$ (1, 4, 2, 3, 4) means that all tasks first execute on P_1, then on P_4, P_2, P_3, and P_4 in this order. We call this sequence the *visit sequence* of the tasks. If an integer l appears more than once in the visit sequence, the corresponding processor P_l is a *reused processor*. In this example P_4 is reused, and each task visits it twice.

The periodic flow-shop model is a generalization of both the traditional flow-shop model and the traditional periodic-job model [7-9]. The periodic *job system* J to be scheduled in a flow shop consists of n independent periodic jobs; each job consists of a periodic sequence of requests for the same computation. In our previous terms, each request is a *task*. The *period* p_i of a job J_i in J is the time interval between the ready times of two consecutive tasks in the job. The deadline of the task in each period is some time Δ after

the ready time of the task. In an m–processor flow shop, each task consists of m subtasks that are to be executed on the m processors in turn following the same order.

3. Flow–Shop Scheduling

Unfortunately, with arbitrary task parameters, the problem of scheduling in a flow shop to meet deadlines is NP–hard [10]. For this reason, we focus on a heuristic algorithm designed for scheduling tasks with arbitrary task parameters. This algorithm, called *Algorithm H*, is described in Figure 1. It makes use of the *effective deadlines* of subtasks. The effective deadline $d_{ij} = d_i - \sum_{k=i+1}^{m} \tau_{ik}$ for the subtask T_{ij} is the point in time by which the execution of the subtask T_{ij} must be completed to allow the later subtasks, and the task T_i, to complete by the deadline d_i. Similarly, the *effective release time* $r_{ij} = r_i + \sum_{k=1}^{j-1} \tau_{ik}$ of a subtask T_{ij} is the earliest point in time at which the subtask can be scheduled.

The effective release times and deadlines of subtasks are computed in Step 1 in Algorithm H. In Step 2, a processor P_b, called the *bottleneck processor*, is identified; a subtask on this processor has the longest processing time $\tau_{\max}$ among all the subtasks on all the processors. Step 3 inflates all the subtasks τ_{ib} on P_b by making their processing times equal to $\tau_{\max}$. Each inflated subtask T_{ib} consists of a busy segment with processing time τ_{ib} and an idle segment with processing time $\tau_{\max} - \tau_{ib}$. Now, all inflated subtasks on P_b have identical processing times. We then use the classical earliest–effective–deadline–first (EEDF) algorithm [11] in Step 4 to schedule the inflated subtasks T_{ib} nonpreemptively on P_b. The resultant schedule S_b is used as a starting point for the construction of the overall schedule S on all the processors. In Step 5, subtasks on the other processors are scheduled in the same order as the subtasks T_{ib} in S_b; hence the resultant schedule is a *permutation schedule*. In this step, every subtask is assigned $\tau_{\max}$ units of time on its processor. Both Step 3 and Step 5 add idle times on all the processors and therefore generate schedules that can be improved. The compaction in Step 6 is a way to improve the performance of Algorithm H (that is, the chance for it to find a feasible schedule when such a schedule exists.) In this step, idle times on all processors are reduced as much as possible.

The classical EEDF algorithm used in Step 4 is priority driven. It assigns priorities to the inflated subtasks T_{ib} according to their effective deadlines: the earlier the deadline, the higher the priority. In particular, we use the more complicated version designed by Garey et al. and described in [11]; it is optimal for nonpreemptive scheduling to meet deadlines of tasks with identical processing times, and with release times and deadlines that are arbitrary rational numbers. This algorithm first identifies forbidden regions in time where tasks are not allowed to start execution and postpones the release times of selected tasks to avoid the forbidden regions. In Figure 1, by release times of T_{ib}, we mean the postponed release times produced by this initial step of the EEDF algorithm from the parameters r_{ib} and d_{ib} as computed in Step 1.

Algorithm H is relatively simple, with complexity $O(n \log n + nm)$. In the special case where the subtasks T_{ij} have identical processing times, that is $\tau_{ij} = \tau_j$ for all $j = 1, 2, \cdots, m$, Step 3 introduces no additional idle time. Furthermore, Step 4 is optimal. If the schedule S_b found by the classical EEDF algorithm is not feasible, then $\{T_{ib}\}$ cannot be feasibly scheduled. Steps 5 and 6 always produce a feasible schedule S of all subtasks on all the processors if the schedule S_b on P_b is feasible. We therefore have the following theorem whose proof follows directly from these arguments [12].

Theorem 1. For nonpreemptive flow–shop scheduling of tasks that have arbitrary release times and deadlines and whose subtasks on each processor P_j have the same processing times τ_j, Algorithm H is optimal.

For the general case of arbitrary processing times τ_{ij}, however, Algorithm H is not optimal. This is caused by the suboptimal scheduling algorithm on a single processor in Step 4 and by the restriction on permutation schedules. Figure 2 shows the results of a simulation experiment to investigate the probability for the Algorithm H to succeed in producing a feasible schedule when feasible schedules exist. In this experiment, Algorithm H was used to schedule task sets which have randomly generated task parameters but are known to have feasible schedules. The fraction of time Algorithm H succeeded in finding a feasible schedule is plotted as a function of m, the number of processors in the flow shop, for different standard deviations (stdv) in the processing times of the subtasks on each processor. We see that when the difference between processing times of subtasks T_{ij} on each processor P_j is small (for example, when the standard deviation is 0.05 or 0.15) Algorithm H performs well. Its performance decreases as the difference in processing times increases.

4. Flow-Shop With Recurrence

The order in which tasks execute on processors in a flow shop with recurrence can also be represented by a *visit graph*, whose set of nodes $\{P_i\}$ represents the processors in the system. There is a directed edge e_{ij} from P_i to P_j with label a if and only if in the visit sequence $V = (v_1, v_2, \cdots, v_a, v_{a+1}, \cdots, v_k)$, $v_a = i$ and $v_{a+1} = j$. A visit sequence can therefore be represented as a path with increasing edge labels in the visit graph. An example of a visit graph for the visit sequence $V = (1, 2, 3, 4, 2, 3, 5)$ is shown in Figure 3. We confine our attention here to a class of visit sequences that contain simple recurrence patterns, called *loops*: some sub-sequence of the visit sequence containing reused processors appears more than once. In the example shown in Figure 3 the labeled path that represents the visit sequence contains a loop. The sub-sequence (2, 3) occurs twice and makes the sequence (4, 2, 3) following the first occurrence of (2, 3) into a loop. In particular, the visit sequence in Figure 3 contains a simple loop which contains no subloop. The *length* of a loop is the number of nodes in the visit graph that are on the cycle. The loop in Figure 3 therefore has length 3.

The problem of scheduling to meet deadlines in a flow shop with recurrence, being a generalization of the flow shop scheduling problem, is NP-hard when task parameters are arbitrary. However, in the special case where (1) all tasks have identical release times, arbitrary deadlines, and the same processing time τ on all processors and (2) the visit sequence contains a single simple loop, there is an optimal, polynomial-time algorithm for scheduling tasks to meet deadlines. This algorithm, called *Algorithm R*, is shown in Figure 4.

Algorithm R is essentially a modified version of the classical EEDF algorithm. We observe that if a loop in the visit graph has length q, the second visit of every task to a reused processor cannot be scheduled before $(q-1)\tau$ time units after the termination of its first visit to the processor. The key strategy used in Algorithm R is based on this observation. Let T_{il} be the subtask at the first visit of T_i to the processor P_{s_l} in the loop of length q, and $T_{i(l+q)}$ be the subtask at the second visit of T_i to the processor. $T_{i(l+q)}$ is dependent on T_{il}. While T_{il} is ready for execution after its release time, $T_{i(l+q)}$ is ready after its release time and after the completion of T_{il}. Step 1 of Algorithm R differs from the classical EEDF algorithm because the effective release times of the second visits are postponed whenever necessary as the first visits are scheduled. Therefore, the optimality of Algorithm R no longer follows in a straightforward way from the optimality of the EEDF Algorithm. Algorithm R is optimal, however, as stated by the following theorem, the proof of which is given in [10].

Theorem 2. For nonpreemptive scheduling of tasks in a flow shop with recurrence, Algorithm R is optimal, when the tasks have identical release times, arbitrary deadlines, identical processing times, and the visit graph contains a single, simple loop.

5. Periodic Flow Shops

To explain a simple method that can be used to schedule jobs in periodic flow shops, we note that each job J_i in a periodic flow-shop job set can be logically divided into m subjobs J_{ij}. The period of J_{ij} is p_i. The subtasks in all periods of J_{ij} are executed on processor P_j and have processing times τ_{ij}. The set of n periodic subjobs $\mathbf{J}_j = \{J_{ij}\}$ is scheduled on the processor P_j. The *total utilization factor* of all the subjobs in $\mathbf{J}_j$ is $u_j = \sum_{i=1}^{n} \tau_{ij}/p_i$. We call the subtask in the kth period of subjob J_{ij} $T_{ij}(k)$. For a given j, the subjobs J_{ij} of different jobs J_i are independent, since the jobs are independent. On the other hand, for a given i, the subjobs J_{ij} of J_i on the different processors are not independent since $T_{ij}(k)$ cannot begin until $T_{i(j-1)}(k)$ is completed. Unfortunately, there are no known polynomial-time optimal algorithms that can be used to schedule dependent periodic jobs to meet deadlines, and there are no heuristic algorithms with known schedulability criteria. Consequently, it is not fruitful to view the subjobs of each job J_i on different processors as dependent subjobs. A more practical approach is to consider all subjobs to be scheduled on all processors as independent periodic subjobs and schedule the subjobs on each processor independently from the subjobs on the other processors. We effectively take into account the dependencies between subjobs of each job in the manner described below.

For the sake of concreteness, let us for the moment give the periodic jobs in our model the following specific interpretation: the consecutive tasks in each job are dependent, that is, the subtask $T_{i1}(k)$ cannot begin until the subtask $T_{im}(k-1)$ is completed. Let γ_i denote the time at which the first task $T_{ij}(1)$ becomes ready. γ_i is also called the *phase* of J_i; it is also the phase γ_{i1} of the subjob J_{i1} of J_i on the first processor. Hence the kth period of the subjob J_{i1} begins at $\gamma_{i1} + (k-1)p_i$. Without loss of generality, suppose that the

set $\mathbf{J}_1$ is scheduled on the first processor P_1 according to the well-known rate-monotone algorithm [7]. Suppose that the total utilization factor u_1 of all the subjobs on P_1 is such that we can be sure that every subtask $T_{i1}(k)$ is completed by the time $\delta_1 p_i$ units after its ready time $\gamma_i = (k-1)p_i$ for some $\delta_1 < 1$. Now, we let the phase of every subjob J_{i2} of J_i on processor P_2 be $\gamma_{i2} = \gamma_{i1} + \delta_1 p_i$. By postponing the ready time of every subtask $T_{i2}(k)$ in every subjob J_{i2} on processor P_2 until its predecessor subtask $T_{i1}(k)$ is surely completed on processor P_1, we can ignore the precedence constraints between subjobs on the two processors. Any schedule produced by scheduling the subjobs on P_1 and P_2 independently in this manner is a schedule that satisfies the precedence constraints between the subjobs J_{i1} and J_{i2}. Similarly, if the total utilisation factor u_2 of all subjobs on P_2 is such that every task in J_{i2} is guaranteed to complete by the time instant $\delta_2 p_i$ units after its ready time, for some $\delta_2 < 1 - \delta_1$, we postpone the phase γ_{i3} of J_{i3} by δ_2, and so on.

Suppose that the total utilization factors u_j for all $j = 1, 2, \cdots, m$ are such that, when the subjobs on each of the m processors are scheduled independently from the subjobs on the other processors according to the rate-monotone algorithm, all subtasks in J_{ij} complete by $\delta_j p_i$ units of time after their respective ready times, for all i and j. Moreover, suppose that each δ_j is a positive fraction, and $\sum_{j=1}^{m} \delta_j \leq 1$. We can postpone the phase of each subjob J_{ij} on P_j by $\delta_j p_i$ units. The resultant schedule is a feasible schedule where all deadlines are met. Given the parameters of $\mathbf{J}$, we can compute the set $\{u_j\}$ and use the existing schedulability bounds given in [8,9] to determine whether there is a set of $\{\delta_j\}$ where $\delta_j > 0$ and $\sum_{j=1}^{m} \delta_j \leq 1$. The job system $\mathbf{J}$ can be feasibly scheduled in the manner described above to meet all deadlines if such a set of δ_j's exists. It is easy to see that with a small modification of the required values of δ_j, this method of scheduling in periodic flow shops can handle the case where the deadline Δ of each task in a job with period p_i is equal to or less than $m\, p_i$ units from its ready time. Similarly, this method can be used when the subjobs are scheduled according to other algorithms, or even different algorithms on different processors, so long as schedulability criteria of the algorithms are known.

6. Summary

We considered here the problem of end-to-end scheduling of tasks with hard deadlines in flow shops. The objective of scheduling is to find a feasible schedule in which all tasks complete by their deadlines whenever feasible schedules exist. The flow-shop model can be used to characterize hard real-time applications on a system containing functionally-dedicated processors and devices. In addition, they also model other types of workload and systems, from data transmissions through a series of links and nodes, to executions of instructions in a pipeline processor, to sequencing of operations in a logic circuit. We have described: (1) a heuristic algorithm for scheduling tasks with arbitrary processing times in flow shops, (2) a polynomial-time optimal algorithm for scheduling task with identical processing times and release times in simple flow shops with recurrence, and (3) an effective method for preemptive scheduling of jobs in periodic flow shops.

Acknowledgement

This work was partially supported by the Navy ONR Contract No. NVY N00014 89-J-1181.

References

[1] Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[2] Lawler, E. L., J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. "Sequencing and scheduling: Algorithms and complexity," Centre for Mathematics and Computer Science, Amsterdam, 1989

[3] Garey, M. R. and D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *J. Assoc. Comput. Mach.* 1976 vol. 23, pp. 461-467.

[4] Garey, M. R., D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop scheduling," *Math. Oper. Res.* 1976 vol. 1, pp. 117-129.

[5] Woodside, C. M. and D. W. Graig, "Local Non-Preemptive Scheduling Policies for Hard Real-Time Distributed Systems," *Proceeding of Real-Time Systems Symposium*, Dec. 1987.

[6] Sha, L., J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proceeding of Real-Time Systems Symposium*, Dec. 1986.

[7] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. Assoc. Comput. Mach.* vol. 20, pp. 46–61, 1973.

[8] Lehoczky, J. P. and L. Sha, "Performance of Real-Time Bus Scheduling Algorithms," *ACM Performance Evaluation Review,* 14, 1986

[9] Peng, D. T. and K. G. Shin, "A New Performance Measure for Scheduling Independent Real-Time Tasks," Technical Report, Department of Electrical Engineering and Computer Science, University of Michigan, 1989.

[10] Bettati, R. and J. W.-S. Liu, "Algorithms for End-to-End Scheduling to Meet Deadlines," Technical Report No. UIUCDCS-R-1594, Department of Computer Science, University of Illinois, 1990.

[11] Garey, M. R., D. S. Johnson, B. B. Simons, and R. E. Tarjan, "Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines," *SIAM J. Comput.* 1981 vol. 10-2, pp. 256–269.

[12] Bettati, R. and J. W.-S. Liu, "Algorithms for End-to-End Scheduling to Meet Deadlines," *Proceeding of the Second IEEE Symposium on Parallel and Distributed Processing,* Dec. 1990.

Algorithm H

Input: Task parameters r_i, d_i and τ_{ij} of **T**.

Output: A feasible schedule of **T**, or "the algorithm fails".

Step 1: Determine the effective release times r_{ij} and effective deadlines d_{ij} of all subtasks.

Step 2: Determine the bottleneck processor P_b on which there is a subtask T_{lb} with the longest processing time τ_{lb}. In other words, for some l, $\tau_{lb} \geq \tau_{ij}$ for all i and j. Let $\tau_{\max} = \tau_{lb}$.

Step 3: Inflate all the subtasks in $\{T_{ib}\}$ by making their processing times equal to $\tau_{\max}$.

Step 4: Schedule the inflated subtasks in $\{T_{ib}\}$ on P_b using the EEDF algorithm. The resultant schedule is S_b.

Step 5: Let t_{ib} be the start time of T_{ib} in S_b on P_b. Propagate the schedule S_b onto the remaining processors as follows: For any task T_i we schedule $T_{i(b+1)}$ to start at time $t_{ib} + \tau_{\max}$ on processor P_{b+1}, $T_{i(b+2)}$ at time $t_{ib} + 2\tau_{\max}$ on P_{b+2}, and so on until T_{im} is scheduled at time $t_{ib} + (m-b)\tau_{\max}$ on P_m. $T_{i(b-1)}$ is scheduled to start at time $t_{ib} - \tau_{i(b-1)}$ on processor P_{b-1}, $T_{i(b-2)}$ at time $t_{ib} - \tau_{\max} - \tau_{i(b-2)}$ on P_{b-2}, and so on, until T_{i1} is scheduled to start at time $t_{ib} - (b-1)\tau_{\max} - \tau_{i1}$ on P_1.

Step 6: Compact the schedule by eliminating as much as possible the lengths of idle periods that were introduced in Step 3 and Step 5. Stop.

Figure 1. Algorithm H for scheduling arbitrary tasks in flow shops.

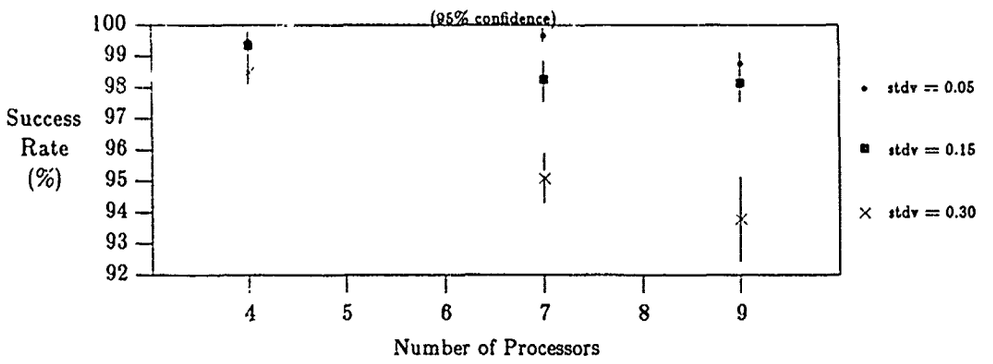


Figure 2. Performance of Algorithm H.

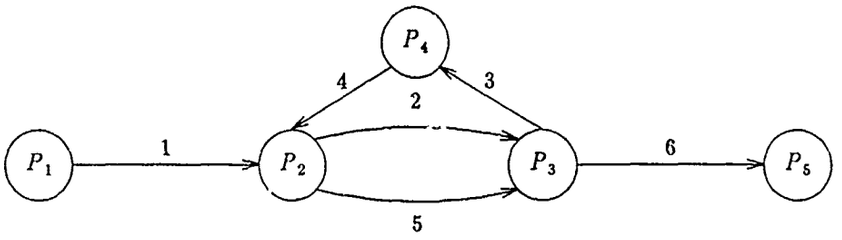


Figure 3. Visit graph for visit sequence $V = (1, 2, 3, 4, 2, 3, 5)$.

Algorithm R:

Input: Task parameters r_{ij}, d_{ij}, τ, and the visit graph G. P_{v_l} is the first processor in the single loop of length q in G.

Output: A feasible schedule S or the conclusion that the tasks in $\{T_i\}$ cannot be feasibly scheduled.

Step 1: Schedule the subtasks in $\{T_{il}\} \bigcup \{T_{i(l+q)}\}$ on the processor P_{v_l} using the modified EEDF algorithm described below: the following actions are taken whenever P_{v_l} becomes idle

(i) If no subtask is ready, leave the processor idle.

(ii) If one or more subtasks are ready for execution, start to execute the one with the earliest effective deadline. When a subtask T_{il} (that is, the first visit of T_i to the processor) is scheduled to start its execution at time t_{il}, set the effective release time of its second visit $T_{i(l+q)}$ to $t_{il} + q\tau$.

Step 2: Let t_{il} and $t_{i(l+q)}$ to be the start times of T_{il} and $T_{i(l+q)}$ in the partial schedule S_R produced in Step 1. Propagate the schedule to the rest of the processors according to the following rules:

(i) If $j < l$, schedule T_{ij} at time $t_{il} - (l-j)\tau$.

(ii) If $l < j \leq l+q$, schedule T_{ij} at time $t_{il} + (j-l)\tau$.

(iii) If $l+q < j < m$, schedule T_{ij} at time $t_{i(l+q)} + (j-l-q)\tau$.

Figure 4. Algorithm R to schedule flow shops with single simple loops.

Real-Time Scheduling of Sensor-Based Control Systems

David B. Stewart and Pradeep K. Khosla

Department of Electrical and Computer Engineering and
The Robotics Institute,
Carnegie Mellon University,
Pittsburgh, PA 15213

Abstract: *Many sensor-based control systems are dynamically changing, and thus require a flexible scheduler. The rate monotonic (RM) real-time scheduling algorithm does not support such dynamic systems very well. On the other hand, with earliest-deadline-first (EDF) and minimum-laxity-first (MLF) dynamic scheduling algorithms, a transient overload in the system may cause a critical task to fail, which is certainly undesirable. This paper proposes a new real-time scheduling algorithm, which we call maximum-urgency-first (MUF), which combines the advantages of the RM, EDF, and MLF algorithms. Like EDF and MLF, MUF has a schedulable bound of 100% for the critical set. And like RM, a critical set can be defined that is guaranteed to meet all its deadlines. The MUF algorithm also allows the scheduler to detect three forms of deadline failures, and call failure handler routines for tasks which fail to meet their deadlines.The MUF scheduler has been implemented as the default scheduler of CHIMERA II, a real-time operating system being used to control sensor-based control systems both at Carnegie Mellon University and elsewhere. There are still many issues to be addressed with regards to the MUF algorithm. This paper also presents those issues, with possible approaches that should be investigated further.*

Keywords: *real-time scheduling, dynamic scheduling, sensor-based control, maximum-urgency-first, rate monotonic algorithm, earliest-deadline-first, minimum-laxity-first, CHIMERA II Real-Time Operating System.*

1 Introduction

Many sensor-based control systems are dynamically changing, and require a flexible scheduler. For example consider the case of a tactile sensor, on the end of a robotic manipulator, that is used to explore an object. Assume the tactile sensor has a resolution of 2^n by 2^m taxels, where n and m can vary dynamically between 1 and 5. When exploring uninteresting parts of an object, such as the straightedge of a table, it is desirable to use the lowest resolution, so that computation time is minimized and sample frequency is fastest, and the robot can follow the edge quickly. As the object becomes more interesting, such as the rounded corner of the table, it is desirable to increase the resolution of the tactile sensor. In doing so, the computational time required to process the data increases, and the frequency of data samples must be decreased (and not necessarily linearly).

The *rate monotonic* (RM) real-time scheduling algorithm does not support such dynamic systems very well. On the other hand, with *earliest-deadline-first* (EDF) and *minimum-laxity-first* (MLF) dynamic scheduling algorithms, a transient overload in the system may cause a critical task to fail, which is certainly undesirable. This paper proposes a new real-time scheduling algorithm, called *maximum-urgency-first* (MUF). It combines the advantages of the RM, EDF, and MLF algorithms. Like EDF and MLF, MUF has a schedulable bound of 100% for the critical set. And like RM, a critical set can be defined that is guaranteed to meet all its deadlines. The MUF algorithm also allows the scheduler to detect three types of timing failures, and call failure handler routines for tasks which fail to meet their deadlines.

Section 2 briefly describes the RM, MLF, and EDF algorithms, and Section 3 describes our new MUF scheduling algorithm. Section 4 describes our implementation of the MUF scheduler as the default scheduler of the CHIMERA II Real Time Operating System[8]. It is being used to control several sensor-based robotic systems at Carnegie Mellon University and elsewhere. The flexibility of the MUF algorithm provides many new possibilities in real-time scheduling of sensor-based control systems. A brief discussion in Section 5 is included to stimulate the reader's interest in the MUF algorithm, and to present a few ideas for further research.We also show that RM, EDF, and MLF are special cases of the MUF algorithm.

2 Related Work

Liu and Layland presented the rate monotonic algorithm as an optimal fixed priority scheduling algorithm, and the earliest-deadline-first and minimum-laxity-first algorithms as optimal dynamic priority scheduling algorithms.[4] Two disjoint scheduling philosophies emerged: static priority scheduling and dynamic priority scheduling. The former consists of using RM, while the latter uses either EDF or MLF as the baseline scheduling algorithm.

2.1 Rate Monotonic Algorithm (RM)

The *rate monotonic algorithm* is a fixed priority scheduling algorithm which consists of assigning the highest priority to the highest frequency tasks in the system, and lowest priority to the lowest frequency tasks. At any time, the scheduler chooses to execute the task with the highest priority. By specifying the period and computational time required by the task, the behavior of the system can be categorized *apriori*.

One problem with the rate monotonic algorithm is that the schedulable bound is less than 100%. The *schedulable bound* of a task set is defined as the maximum *CPU utilization* for which the set of tasks can be guaranteed to meet their deadlines. The CPU utilization of task P_i is computed as the ratio of worst-case computing time C_i to the period T_i. The total utilization U_n for n tasks is calculated as follows:

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1}$$

For the RM algorithm, the worst-case schedulable bound W_n for n tasks is

$$W_n = n(2^{1/n} - 1) \tag{2}$$

From (2), $W_1 = 100\%$, $W_2 = 83\%$, $W_3 = 78\%$, and in the limit, $W_\infty = 69\%$ ($ln\ 2$). Thus a set of tasks for which total CPU utilization is less than 69% will always meet all deadlines. All tasks will be guaranteed to meet their deadlines if $U_n \le W_n$. If $U_n > W_n$, then the subset of highest-priority tasks S such that $U_s \le W_s$ will be guaranteed to meet all deadlines, and will thus form the *critical set*. Note that the worst case values are pessimistic, and it has been shown that for the average case $W_\infty = 88\%$[3].

Another problem with RM is that it does not support dynamically changing periods very well, a feature required by some sensor-based control systems. For example, a task set with three tasks P_1, P_2, and P_3, of periods $T_1 = 30$ms, $T_2 = 50$ms, and $T_3 = 100$ms would have the following fixed priority assignment (from highest to lowest): P_1, P_2, P_3. Suppose the period of P_1 changes to $T_1 = 75$ms. Under the RM algorithm, we would require that the priorities of each task be reassigned to the ordering P_2, P_1, P_3, which violates the condition that priorities are static.

The problems with RM encourage the use of dynamic priority algorithms. Although many such algorithms exist, we restrict our attention in this paper to EDF and MLF.

2.2 Earliest-Deadline-First Scheduling Algorithm (EDF)

As the name implies, the *earliest-deadline-first* algorithm uses the deadline of a task as its priority. The task with the earliest deadline has the highest priority, while the task with the latest deadline has the lowest priority. One advantage of this algorithm is that the schedulable bound is 100% for all task sets. Secondly, because priorities are dynamic, the periods of tasks can be changed at any time.

A major problem with the EDF algorithm is that there is no way to guarantee which tasks will fail in a *transient overload* situation. In many systems, although the average case utilization is less than 100%, it is possible that the worst-case utilization is above 100%, leaving the possibility of one or more tasks failing. In such cases, it is desirable to control which tasks fail and which succeed during such a transient overload. In the RM algorithm, low priority tasks will always be the first to fail. However, no such priority assignment exists with EDF, and thus there is no control of which task fails during a transient overload. As a result, it is possible that a very critical task may fail at the expense of a lesser important task.

2.3 Minimum-Laxity-First Scheduling Algorithm (MLF)

Our purpose in describing the *minimum-laxity-first* algorithm in this section is not to compare it to RM or EDF, but rather to introduce it as a basis for the *maximum-urgency-first* algorithm proposed in this paper. The minimum-laxity-

first algorithm assigns a *laxity* to each task in a system, then selects the task with the minimum laxity to execute next. Laxity is defined as follows:

$$\text{laxity} = \text{deadline\_time} - \text{current\_time} - \text{CPU\_time\_still\_needed} \quad (3)$$

Laxity is a measure of the flexibility available for scheduling a task. A laxity of t_l means that even if the task is delayed by t_l time units, it will still meet its deadline. A laxity of zero means that the task must begin to execute *now* or it will risk failing to meet its deadline.

The main difference between MLF and EDF is that MLF takes into consideration the execution time of a task, which EDF does not do. Like the earliest-deadline-first algorithm, MLF has a 100% schedulable bound, but there is no way to control which are guaranteed to execute during a transient overload. In the next section, we present the MUF algorithm, which allows the control of task failures during transient overload, while maintaining the flexibility of a dynamic scheduler, and 100% schedulable bound for the critical set.

3 Maximum-Urgency-First Algorithm (MUF)

The *maximum-urgency-first* scheduling algorithm which we have developed is a combination of fixed and dynamic priority scheduling, also called *mixed priority* scheduling. With this algorithm, each task is given an *urgency*. The urgency of a task is defined as a combination of two fixed priorities, and a dynamic priority. One of the fixed priorities, called the *criticality*, has higher precedence over the dynamic priority. The other fixed priority, which we call *user priority*, has lower precedence than the dynamic priority. The dynamic priority is inversely proportional to the laxity of a task.

The MUF algorithm consists of two parts. The first part is the assignment of the criticality and user priority, which is done *apriori*. The second part involves the actions of the *MUF scheduler* during run-time

The steps in assigning the criticality and user priority are the following:

1. As with RM, order the tasks from shortest period to longest period.
2. Define the critical set as the first N tasks such that the total worst-case CPU utilization does not exceed 100%. These will be the tasks that do not fail, even during a transient overload of the system. If a critical task does not fall within the critical set, then *period transformation*, as used with RM,[6] can also be used here.
3. Assign *high* criticality to all tasks in the critical set, and *low* criticality to all other tasks.
4. Optionally assign a unique user priority to every task in the system.

The static priorities are defined once, and do not change during execution. The dynamic priority of each task is assigned at run-time, inversely proportional to the laxity of the task. Before its cycle, each task must specify its desired start time, deadline time, and worst-case execution time.

Whenever a task is added to the ready queue, a reschedule operation is performed. The MUF scheduler is used to determine which task is to be selected for execution, using the following algorithm:

1. Select the task with the highest criticalness.
2. If two or more tasks share highest criticalness, then select the task with the highest dynamic priority (i.e. minimum laxity). Only tasks with pending deadlines have a non-zero dynamic priority. Tasks with no deadlines have a dynamic priority of zero.
3. If two or more tasks share highest criticalness, and have equal dynamic priority, then the task among them with the highest user priority is selected.
4. If there are still two or more tasks that share highest criticalness, dynamic priority, and highest user priority, then they are serviced in a *first-come-first-serve* manner.

The optional assignment of unique user priorities for each task ensures that the scheduler never reaches step 4., thus providing a deterministic scheduling algorithm. We have yet to investigate the best method for assigning the user priorities.

To demonstrate the advantage of MUF over RM and EDF, consider the task set shown in Figure 1. We assume that the deadline of each task is the beginning of the next cycle. Four tasks are defined, with a total worst-case utilization of over 100%, thus in the worst-case, missed deadlines are inevitable. Figure 1(a) shows the schedule produced by a static priority scheduler when priorities are assigned using the RM algorithm. In this case, only P_1 and P_2 are in the critical set, and are guaranteed not to miss deadlines. Expectably, both P_3 and P_4 miss their deadlines. When using the EDF algorithm, as in Figure 1(b), tasks P_1 and P_2 fail. However, any task may have failed, since with EDF there is no way to predict the failure of tasks during a transient overload of the system.

With the MUF algorithm, all tasks in the critical set are guaranteed not to miss deadlines. In our example, the combined worst-case utilization of P_1, P_2, and P_3 is less than 100%, and thus they form the critical set. Only task P_4 can miss deadlines, because it is not in the critical set. Figure 1(c) shows the schedule produced by the MUF scheduler. Note the improvement over RM: because of a higher schedulable bound for the critical set, task P_3 is also in the critical set and thus does not miss any deadlines. Also, unlike EDF, we are able to control that the only task that may fail is P_4.

The choice of using MLF to calculate the dynamic priority instead of EDF enables the scheduler to detect missed deadlines. There are three failures which the MUF scheduler can detect:

1. A task has not completed its cycle when the deadline time has been reached;
2. A task was given as much CPU time as was requested in the worst-case, yet it still did not meet its deadline;
3. The task will not meet its deadline because the minimum CPU time requested cannot be granted. This case also requires that the minimum amount of CPU time required by a task is specified.

The first case is the standard notion of a missed deadline. The second case will detect bad worst-case estimates of execution time. The third case allows the MUF scheduler to make the most of its CPU time, and it will not start executing a task if that task has no possibility to finish before its deadline, thus providing the early detection of missed deadlines. Instead, the CPU time can be reclaimed for ensuring that other tasks do not miss deadlines, or to call alternate, shorter threads of execution.

4 Implementation

One concern of the MUF scheduler is the overhead that would be required during each reschedule operation. The overhead of the MUF scheduler can be reduced by encoding the algorithm into a single *urgency* value, hence the name of the algorithm. Figure 2 shows an n-bit urgency value, which was encoded using c bits for criticality, d bits for the dynamic priority, and u bits for the user priority. With such an encoding, the range of criticalities, dynamic priorities, and user priorities are 0 to 2^c-1, 0 to 2^d-1, and 0 to 2^u-1 respectively. The MUF scheduler must then only calculate a single dynamic priority for each task, then select the task with the maximum urgency. This encoding scheme can be used to implement the MUF algorithm as long as c, d, and u are all greater than or equal to $log_2(max\ number\ of\ tasks\ in\ system)$. Such encoding allows the maximum urgency scheduler to be implemented efficiently.

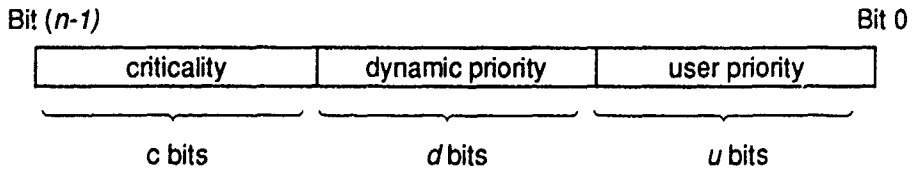


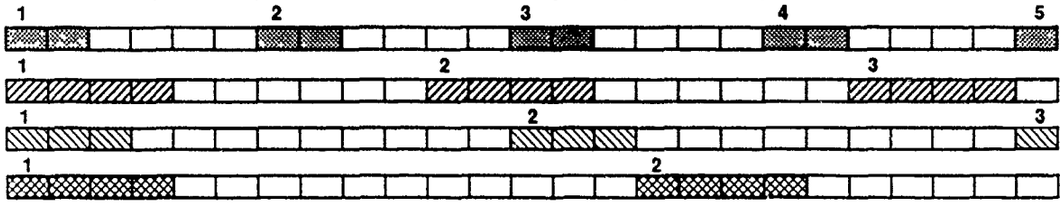
Figure 2: Encoded n-bit *Urgency* Value

We have implemented the MUF scheduler as the default scheduler of the CHIMERA II Real-Time Operating System [8]. CHIMERA II is being used both at Carnegie Mellon University and elsewhere, on a variety of sensor-based control systems, including the CMU Direct Drive Arm II [2] and the CMU Reconfigurable Modular Manipulator System [5].

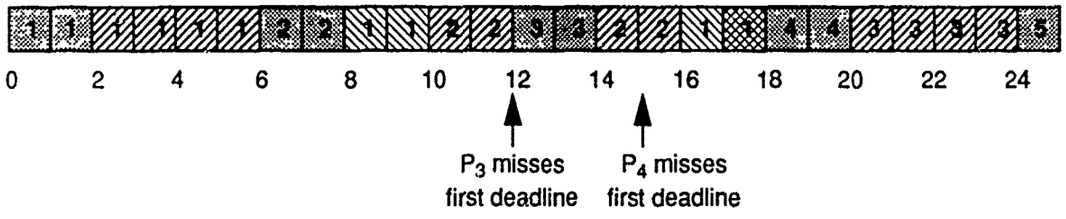
On an Ironics IV3220 Single Board Computer, with a 20 MHz M68020 processor, a reschedule operation with four ready tasks (excluding context switch time), takes 28 microseconds. The context switch takes another

Task	Priority(RM)	Criticality(MUF)	Period	CPU time	Utilization	Legend
P_1	High	High	6	2	33%	
P_2	Med High	High	10	4	40%	
P_3	Med Low	High	12	3	25%	
P_4	Low	Low	15	4	27%	

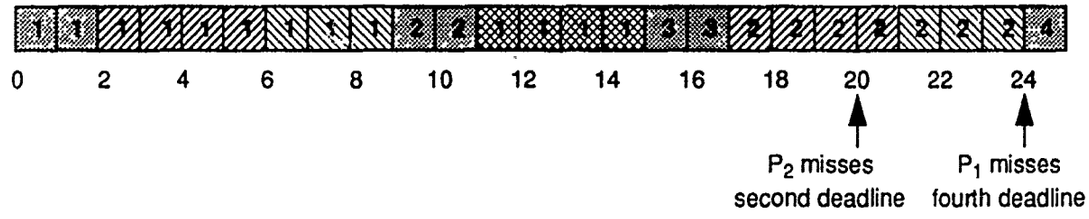
CPU time requested by each task (deadline is beginning of following cycle):



(a) Schedule generated when using *Rate Monotonic* algorithm:



(b) Schedule generated when using *Earliest-Deadline-First* algorithm:



(c) Schedule generated when using *Maximum-Urgency-First* algorithm:

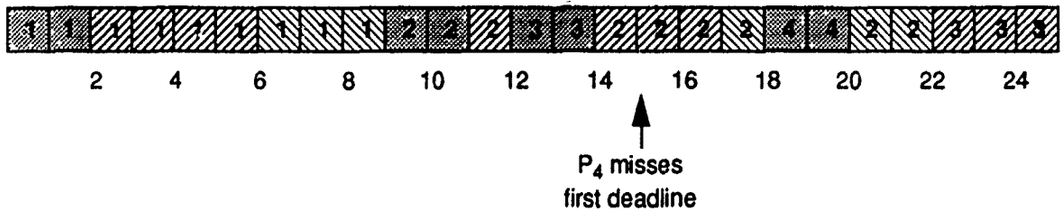


Figure 1: Example comparing RM, EDF, and MUF algorithms

66 microseconds, for a total of 94 microseconds. With a 1 millisecond clock, we maintain over 90% CPU utilization, while with a 10 millisecond clock we maintain over 98% utilization. This type of performance allows the scheduler to be used with sensor-based control applications that have tasks with frequencies as high as 1000 Hz.

Our implementation also offers deadline failure handling. Whenever a task fails to meet its deadline, an optional failure handler is called on behalf of the failing task. The failure handler can be programmed to execute either at the same or different criticality and user priority than the failing task. Such functionality is essential in predictable and fault-tolerant systems. Much emphasis in hard real-time systems has gone into ensuring that critical tasks always meet their deadlines. However, very little has been said about what to do about those tasks which fail to meet their deadlines during a transient overload. Possible actions include the following: aborting the task and preparing it to restart the next period; sending a message to some other part of the system to handle the error; modifying the priority of the task, and continuing its execution; performing emergency handling, such as a graceful shutdown of part of the system or sounding an alarm; maintaining statistics on failure frequency to aid in analyzing the system; in the case of iterative algorithms, returning the current approximate value regardless of precision. Any of these actions and other user-defined actions can be implemented using the deadline failure handling available with our MUF scheduler.

Estimating the execution time of tasks is often difficult. For example, most commercially-available hardware is geared towards increasing average performance via the use of caches and pipelines. Such hardware is often used to implement real-time systems. As a result, the execution time cannot necessarily be predicted accurately. Under-estimating worst-case execution times can create serious problems, as it is possible that a task in the critical set also fails. The use of deadline failure handlers is thus recommended for all tasks in a system, and not only those tasks which are not guaranteed. Our MUF scheduler provides this ability.

5 Discussion of MUF Algorithm

There are still many issues to be addressed with regards to the MUF algorithm. This section presents those issues, with possible approaches, which should be investigated further.

Aperiodic Events: The presentation of the MUF algorithm in this paper assumed only periodic tasks. Most real-time systems also have aperiodic events. Because MUF is a dynamic scheduler, aperiodic events can readily be included in the system without changing the basic MUF scheduler. However, such events must not cause tasks from the critical set to fail. Several methods have been adopted with the RM algorithm, including the *sporadic server* [7]. Similar methods can possibly be used with the MUF algorithm. For example, an aperiodic server can be given a criticality higher than the critical set. Its CPU utilization is included in the computation of the critical set, and calculated such that no critical tasks will miss deadlines if the aperiodic server does not use more CPU time than it is allotted. As with any periodic task, a deadline and maximum execution time is specified. If the server uses up all its time, then the failure handler is called, which replenishes the server's execution time, or blocks the server until its CPU time can be safely replenished

Task Synchronization: Real-time tasks are usually not independent. The sharing of limited resources, and the communication between tasks require appropriate synchronization or scheduling. With the RM algorithm, *priority ceiling protocol* [6] semaphores are often used for ensuring critical tasks still meet their deadlines in the presence of task dependencies. For the dynamic scheduling algorithms, both *dynamic priority ceiling protocol* semaphores [1] and resource scheduling [9] have been proposed. Adaptation of one or more of these methods to the MUF algorithm may be possible.

Varying Time Constraints: In the introduction of this paper we gave an example of dynamically changing timing constraints that may be encountered in sensor-base control systems. The MUF algorithm supports such tasks. Because the MLF algorithm is used to schedule tasks within the critical set, their frequencies and worst-case execution times can change dynamically. In order to guarantee tasks in the critical set in a dynamically changing environment, the worst-case utilization U_P for every task P is defined as $U_P = \max(C_{Pc}/T_{Pc})$, which is the maximum utilization of task P during any one cycle. Any combination of period and CPU execution time can then be used, as long as $C_{Pc}/T_{Pc} \leq U_P$ for every cycle Pc. This is a signficant improvement over RM, where a change in period and CPU execution time may cause the critical set to change, even though utilization remains constant. When defining the MUF algorithm in Section 3, we first ordered tasks from shortest to longest period. This step can be relaxed, and MUF will still perform properly, but at the cost of non-critical tasks possibly failing unnecessarily.

Modular Design: In developing modular systems, it may be desirable to specify timing constraints on a per-module instead of per-task basis. For example, a module may consist of two dependent tasks, such that the combined worst-case CPU utilization is less than the sum of the utilization of the two tasks. In assigning priorities using RM, the frequency of the tasks plays an important role. However, with the MUF algorithm, only the utilization plays a role. By taking advantage of combined utilizations, it is possible to have a critical set in which the sum of the utilizations of all tasks within the set is *over* 100%, but the worst-case utilization for any one time slice is still less than 100%.

RM, EDF, and MLF as Special Cases of MUF: Without any modification, the MUF scheduler can also be used to schedule task sets using either the RM, EDF, or MLF algorithm. For example, instead of assigning criticalities according to the MUF algorithm, assign criticalities to tasks in the same way as priorities are assigned using the RM algorithm. Every task thus has a different criticality, and MUF behaves as a static highest priority scheduler. Deadline and execution times can still be specified to the MUF scheduler, even though they will not be used in the selection of which task to execute. This allows the MUF scheduler to still detect deadline failures, even though the RM priority assignment is used. Most fixed priority schedulers do not have such capabilities. If all tasks are given the same criticality, then the MUF scheduler behaves as an MLF scheduler. If the tasks all specify *zero* as the worst-case execution time, then the MUF scheduler reduces to an EDF scheduler, since the urgency of the task reduces to a function of deadline time. Note that in the latter case, early detection of deadline failures and failures due to under-estimating worst-case execution times cannot be detected.

6 Acknowledgments

The research reported in this paper is supported, in part, by U.S. Army AMCOM and DARPA under contract DAAA-2189-C-0001, by the Department of Electrical and Computer Engineering, and by The Robotics Institute at Carnegie Mellon University. Partial support for David B. Stewart is provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Graduate Schclarship. Special thanks also goes to Donald E. Schmitz, with whom numerous discussions eventually led to the development of some of the ideas presented in this paper.

7 References

[1] Chen, M.-I., and K. J. Lin, "Dynamic Priority Ceilings: a Concurrency Control Protocol for Real-Time Systems," Univ. Illinois at Urbana-Champaign, IL, Tech Report UIUCDCS-R-89-1511, April 1989.

[2] Kanade, T., P.K Khosla, and N. Tanaka, "Real-Time Control of the CMU Direct Drive Arm II Using Customized Inverse Dynamics," in *Proceedings of the 23rd IEEE Conference on Decision and Control*, Las Vegas, NV, December 1984, pp. 1345-1352.

[3] Lehoczky, J., L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings 10th IEEE Real-Time Systems Symposium*, Santa Monica, CA, December 1989, pp. 166-171.

[4] Liu, C. L., and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *Journal of the Association for Computing Machinery*, v.20, n.1, January 1973, pp. 44-61.

[5] Schmitz, D. E., P. K. Khosla, and T. Kanade, "The CMU Reconfigurable Modular Manipulator System," in *Proceedings of the International Symposium and Exposition on Robots* (designated 19th ISIR), Sydney, Australia, Nov. 1988, pp. 473-488.

[6] Sha, L., J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemtive Scheduling," in *Proceedings 10th IEEE Real-Time Systems Symposium*, Santa Monica, CA, December 1989.

[7] Sprunt, B., L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Journal of Real-Time Systems*, v.1, n.1, Nov 1989, pp. 27-60.

[8] Stewart, D. B., D. E. Schmitz, and P. K. Khosla, "Implementing Real-Time Robotic Systems using CHIMERA II," in *Proceedings of 1990 IEEE International Conference on Robotics and Automation*, Cincinnatti, OH, May 1990, pp. 598-603.

[9] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, v.SE-13, n.5, May 1987, pp. 564-577.

EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE
(in conjunction with)
17th IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING

CALL FOR PAPERS

May 15 – 17, 1991
Atlanta, GA, USA

WORKSHOP CHAIRS

Krithi Ramamritham
Dept. of Computer & Info. Science
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003
USA
413 545-0196
krithi@nirvan.cs.umass.edu

Wolfgang A. Halang
Dept. of Computing Science
University of Groningen
P. O. Box 800
NL-9700 AV Groningen
The Netherlands
+31-50-63 39 39
halang@cs.rug.nl

This workshop has several goals:

- to investigate advances in *real-time* operating systems, software, and programming languages,
- to promote interaction among researchers and practitioners,
- to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

Workshop attendees will explore current ideas on real-time software, programming languages, and operating systems. Position papers describing new ideas, promising approaches, and work in progress are considered particularly appropriate.

Possible topics of this workshop include:

- Real-time operating systems,
- Real-time programming, requirements analysis and specification,
- Evaluation of real-time systems,
- Real-time scheduling and resource management,
- Examples of real-time (control) systems with challenging time constraints.

Prospective attendees should send 10 copies of a (no more than) 5-page position paper to *Krithi Ramamritham* by January 15, 1991. To facilitate the reviewing process, it is recommended that submission also be sent electronically—in the form of plain ASCII files. The position paper should focus on insights and lessons gained from recent research and practical experience in real-time operating systems and software. Complete details regarding the workshop will be sent to all participants along with acceptance letters by March 15, 1991. Preprints of the accepted papers will be made available at the Workshop. Proceedings will be published after the workshop by Pergamon Press in the IFAC Proceedings Series. Attendance will be limited to approximately 75 active workers in the field.

Co-sponsored by:
- IEEE Computer Society
  Technical Committee on Real-Time Systems
- Office of Naval Research

IEEE

In cooperation with:
- IFAC
  Technical Committee on Computers
  Working Group on Real-Time Programming

and
- IFIP Working Group 5.4
  on Computerized Process Control

ADVANCE REGISTRATION

Joint
Eighth IEEE Workshop on Real-Time Operating Systems and Software
and
17th IFAC Workshop on Real-Time Programming Languages

May 15-17, 1991
Atlanta, GA, USA

Tel. 413-545-0196 Fax: 413-545-1249

NAME:
AFFILIATION:
ADDRESS:
CITY/STATE/ZIP:
ELECTRONIC MAIL ADDRESS:
PHONE/FAX:

MEALS: Any dietary restrictions, i.e., vegetarian or other?

ADVANCE REGISTRATION FEES:

	Before April 15	After Arpil 15
IEEE Members	$130	$150
Non-Members	$155	$185
Student	$ 55	$ 55

IEEE Membership #:_____

Proceedings and social functions are included in the price.

******

After completing this form, return with check (U.S. Dollars) made payable to Eighth IEEE Workshop on Real-Time to:

Ms. Betty Hardy
University of Massachusetts
Computer and Information Science Dept.
Lederle Graduate Research Center, A243
Amherst, MA 01003 USA

NOTE: Additional information regarding hotel and airlines is attached.

Final Program

JOINT
IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE
IFAC WORKSHOP ON REAL-TIME PROGRAMMING

MAY 15-17, 1991
Atlanta, GA

Wednesday, May 15, 1991

• LUNCHEON	for Speakers and Panelists (Hotel Restaurant)	12:00 - 1:15
•WELCOME	Krithi Ramamritham Wolfgang Halang Karsten Schwan	1:30 - 1:35
SESSION 1	OPERATING SYSTEMS *Chair: Kwei-Jay Lin* University of Illinois	1:35 - 3:15

1. *Multiprocessor Synchronization Primitives with Priorities*
   Evangelos P. Markatos, University of Rochester, New York
2. *YARTOS: Kernel Support for Efficient, Predictable Real-Time Systems*
   Kevin Jeffay, Don Stone, Dan Poirier
   University of North Carolina at Chapel Hill, North Carolina
3. *Dynamic Scheduling for Hard Real-Time Systems: Toward Real-Time Threads*
   Hongyi Zhou, Karsten Schwan, Georgia Institute of Technology, Atlanta
4. *A Reliable Multicast Protocol for Distributed Real-Time Systems*
   H. Kopetz, G. Grünsteidl, Technical University of Vienna, Austria

• BREAK		3:15 - 3:45
• PANEL DISCUSSION:	"Operating Systems: Interfaces/Standards"	3:45 - 5:00

C. Douglass Locke, IBM (Chair)
Hide Tokuda, Carnegie Mellon University
Karen Gordon, IDA/CSED
Robert Cook, University of Virginia

• RECEPTION	Evergreen Conference Center (Busses leave hotel at 6:30)	7:30 - 10:30

Thursday, May 16, 1991 - Morning

• **BREAKFAST** for Speakers and Panelists (Hotel Restaurant) 7:00 - 8:15

SESSION 2 **DESIGN OF REAL-TIME SYSTEMS** 8:30 - 10:15
Chair: Juan de la Puente
Universidad Politenica de Madrid

1. *GARTEN: A Programming Environment for Real-Time Software Development*
   Keith Ranson, C. Marlin, Wei Zhao
   The University of Adelaide, South Australia and
   Texas A&M University, College Station

2. *Schedulability, Program Transformations and Real-Time Programming*
   Alexander D. Stoyenko, Thomas J. Marlowe
   New Jersey Institute of Technology, Newark and
   Seton Hall University, South Orange, New Jersey

3. *PIPS: An Integrated Approach to the Design of Real-Time Systems*
   Chien-Chung Shen, Rajive Bagrodia
   University of California, Los Angeles

4. *Graphical Prototyping of Tasking Behaviour*
   R. Lintulampi, P. Pulli
   Technical Research Centre of Finland, Oulu

• **BREAK** 10:15 - 10:45

• **PANEL DISCUSSION:** **"Languages: Ada? object-oriented ?"** 10:45 - 12:00

Ted Baker, Forida State University (Chair)
N. Natarajan, Pennsylvania State University
Wolfgang Halang, Groningen University
Offer Pazy, Intermetrics, Inc.

Thursday, May 16, 1991 - Afternoon

• **LUNCHEON** Poolside, weather permitting 12:00 - 1:30

SESSION 3 **APPLICATIONS/EXPERIENCE** 1:30 - 3:15
Chair: Andre van Tilborg
Office of Naval Research

1. *Application of Real-Time Scheduling Theory to Multiprocessor Pipelines*
   Robert J. Fornaro, William D. Allen,
   North Carolina State University, Raleigh

2. *Computer Music Performance as a Real-Time Testbed*
   David H. Jameson, IBM T.J. Watson Research Center,
   Yorktown Heights, New York

3. *Specifying Hard Real-Time Software: Experience with a Language and a Verifier*
   Constance Heitmeyer, Bruce Labaw,
   Naval Research Laboratory, Washington, DC

4. *Designing a Hard Real-Time System with Automatic Memory Management*
   Edward E. Ferguson, Dexter S. Cook, David H. Bartley,
   Texas Instruments Inc., Dallas

• **BREAK** 3:15 - 3:45

Thursday, May 16, 1991- Afternoon

SESSION 4 — TIMING–ANALYSIS/MONITORING — 3:45 - 5:30

Chair: Al Mok
University of Texas at Austin

1. *Application of Partial Evaluation to Hard Real-Time Programming*
   Vivek Nirkhe, William Pugh, University of Maryland, College Park
2. *Predictable Real-Time Caching in the Spring System*
   Douglas Niehaus, Erich Nahum, John A. Stankovic
   University of Massachusetts, Amherst
3. *Static Analysis of Timing Properties for Distributed Real-Time Programs*
   Horst F. Wedde, Bogdan Korel, Dorota M. Huizinga
   Wayne State University, Detroit, Michigan
4. *An Integrated Approach to Monitoring and Scheduling in Real-Time Systems*
   Farnam Jahanian, Ragunathan Rajkumar
   IBM T.J. Watson Research Center, Yorktown Heights, New York

- **"Birds of a Feather" Sessions** — 8:00 - ?

Friday, May 17, 1991- Morning

• **BREAKFAST**	for Speakers and Panelists (Hotel Restaurant)	7:00 - 8:15
SESSION 5	**POT POURRI** *Chair: Insup Lee* University of Pennsylvania	8:30 - 10:15

1. *New Paradigms for Real-Time Database Systems*
   Robert P. Cook, Sang H. Son, Henry Y. Oh, Juhnyoung Lee,
   University of Virginia, Charlottesville

2. *Generating Synthetic Workloads for Real-Time Systems*
   Daniel L. Kiskis, Kang G. Shin, The University of Michigan, Ann Arbor

3. *Managing Beliefs, Desires, and Time in Real-Time Systems*
   Tom Bihari, Prabha Gopinath, Tom Walliser
   Adaptive Machine Technologies, Columbus, Ohio and
   North American Philips Corp., Briarcliff Manor, New York

4. *Adding Problem-Solving Capabilities to Existing Real-Time Systems*
   C.J. Paul, Anurag Acharya, Bryan Black, Jay Strosnider
   Carnegie Mellon University, Pittsburgh, Pennsylvania

• **BREAK**		10:15 - 10:45
• **PANEL DISCUSSION:**	**"Fault-Tolerance and Real-Time Systems"**	10:45 - 12:00

Kang Shin, University of Michigan (Chair)
Farnam Jahanian, IBM
Jay Strosnider, CMU
Gary Koob, Office of Naval Research

Friday, May 17, 1991 - Afternoon

• **LUNCHEON**	Poolside, weather permitting	12:00 - 1:30

SESSION 6 — **SCHEDULING POT POURRI** — 1:30 - 3:15

John Lehoczky, Chair
Carnegie Mellon University

1. *Limitations Concerning On-Line Scheduling Algorithms for Overloaded Real-Time Systems*
   Sanjoy K. Baruah, Louis E. Rosier, University of Texas, Austin
2. *Hard Real-Time Scheduling: The Deadline-Monotonic Approach*
   N.C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings
   University of York, England
3. *Algorithms for Flow-Shop Scheduling to Meet Deadlines*
   R. Bettati, Jane W.S. Liu, University of Illinois, Urbana
4. *Real-Time Scheduling of Sensor-Based Control Systems*
   David B. Stewart, Pradeep K. Khosla
   Carnegie Mellon University, Pittsburgh, Pennsylvania and
   The Robotics Institute, Pittsburgh

• **BREAK**		3:15 - 3:45
• **PANEL DISCUSSION:**	**"Who needs one more scheduling algorithm (for yet another task model)?"**	3:45 - 5:00

Jane Liu, University of Illinois (Chair)
Jack Stankovic, University of Massachusetts
Karsten Schwan, Georgia Institute of Technology
Marc Donner, IBM
Ed Ferguson, TI

READ ME FIRST

This READ ME FIRST gives you organizational information about the workshop.

Enclosures At the time of registration you should receive the following items.

1. Conference Proceedings (separate).
2. Envelope (large) containing all other items except for the conference proceedings.
3. READ ME FIRST (this sheet).
4. Badge (clip-on style) with your name and affiliation. The name tag will be printed if you are preregistered or lucky. Otherwise it will be written by hand when you check in. Please wear your badge during the conference.
5. Speaker ribbon (blue) (only if you are presenting a paper). Please claim a blue speaker ribbon if you should have one.
6. Program Committee Member ribbon (red) (only if you are a program committee member). Please claim a red program committee member ribbon if you should have one.
7. Final Program of the workshop (7 pages) in protective transparent binder.
8. Envelope (small) with meal tickets:
   (a) Pink: Wednesday night reception and bus ticket.
   (b) Yellow: Thursday lunch.
   (c) Blue: Friday lunch.
9. Stone Mountain Park Brochure.
10. Buckhead Brochure.

Wednesday Night Reception/Laser Light Show The Wednesday night reception will be held at the Evergreen Conference Center in Stone Mountain Park with the following time schedule:

- 6:30 pm Chartered bus departs from Sheraton Century Center hotel. Please use the pink ticket as bus ticket and keep it for later use at the reception.
- 7:30 pm Reception at the Evergreen Conference Center. Reuse the pink ticket as your dinner ticket.
- 9:00 pm Departure of bus from Evergreen Conference Center to the Laser Light Show on *the other side* of Stone Mountain Park. Please don't leave anything behind at the Evergreen Conference Center because the bus will not return to it. In case of rain the laser light show will be cancelled and the bus will return directly to the Sheraton Century Center hotel at this time. Do not plan on walking from one side of the park to the other because the park is huge.
- 9:30 pm Laser Light Show. The daily laser light show at Stone Mountain Park is an open air event. You will either stand or sit on the grass. You may want to bring a towel from the hotel to sit on.
- 10:30 pm Departure of bus back to the Sheraton Century Center hotel.

If you choose to drive to Stone Mountain Park yourself, be prepared to pay an entrance fee at the gate (it is free if you go by bus) and follow the following directions:

- From the Sheraton Century Center hotel take I-85 North to I-285.
- Take I-285 East to Exit 30B (Hwy 78/Stone Mountain Pkwy).
- Go east (no choice hear) on Stone Mountain Pkwy for 7.5 miles to the East Gate of Stone Mountain Park.
- Inside the park, turn left at the first opportunity.
- Go for 2 miles past the beach, golf course, and campground. The Evergreen Conference Center is on your right directly after the dam.

Dietary Restrictions If you are a vegetarian, please say so when you check in. We will try to accomodate you as best we can. We need to give the hotel 24h notice of the number of vegetarian meals.

Address List Address lists of the conference attendees will be available at the registration desk for unattended pickup on the last day of the workshop.

Message Board There will be a message board at the registration desk for general information and personal messages. To leave a personal message, write it on a 3"x5" index card (provided at the desk). Please include "to:" and "from:" fields. A helper will put the name of the addressee on the board.

Telephone Unfortunately, we cannot provide a telephone for incoming messages. The main number of the Sheraton Century Center hotel is 325-0000. The hotel has payphones for outgoing calls.

Attendee List

Joint
Eight IEEE Workshop on Real-Time Operating Systems and Software
and
17th IFAC Workshop on Real-Time Programming Languages

May 15-17, 1991
Atlanta, GA, USA

Allen, William D., North Carolina State Univ., Precision Engin. Center, Campus Box 7918, Raleigh, NC 27695-7918, USA, allen@pevsa.nesu.edu, Tel: 9 .9 737-3096.

Audsley, Neil C., University of York, Dept. of Computer Science, York, England, neil@uk.ac.york.minster, Tel: +904-432-761.

Bagrodia, Rajive, UCLA, 3531F Boelter Hall, Los Angeles, CA 90024, USA, rajive@cs.ucla.edu, Tel:213 825-0956, Fax: 213 825-2273.

Baker, Ted, Florida State Univ., Dept. of Computer Science B-173, Tallahassee, Fl 32306-4019, USA, baker@nu.cs.fsu.edu, Tel: 904 644-5452, Fax: 904 644-0058.

Baruah, Sanjoy K., Dept. of Computer Sciences, The Univ. of Texas at Austin, Austin, TX 78712-1188, USA, sanjoy@cs.utexas.edu, Tel: 512 471-9588, Fax: 512 471-0616.

Bettati, Riccardo, Univ. of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, Il 61801, USA, bettati@cs.uiuc.edu, Tel: 217 244-0432.

Bihari, Tom, Adaptive Machine Technologies, 1218 Kinnear Road, Columbus, Ohio 43212, USA, amt@eagle.eng.ohio-state.edu, Tel: 614 486-7741.

Brockmann, Uwe, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, uwe@cc.gatech.edu, Tel: 404 894-3982, Fax: 404 853-9378.

Chronaki, Catherine, Computer Science Dept., University of Rochester, Rochester, NY 14627, USA, chronaki@cs.rochester.edu, Tel: 716 275-7230.

Cook, Robert P., Univ. of Virginia, Thornton Hall, Charlottesville, VA 22903, USA, cook@cs.virginia.edu, Tel: 804 982-2215.

Puente, Juan A. de la, Grupo de Ingeneria de Control, ETSI Telecomunicacion, Ciudad Universitaria, E-28040 Madrid, Spain, jpuente@dit.upm.es, Tel:(34-1)3367342, Fax: (34-1)5432077.

Donner, Marc, IBM Research, P.O. Box 218, Yorktown Heights, NY 10598, USA, donner@watson.ibm.com, Tel: 914 945-1234.

Ferguson, Edward E., Texas Instruments, PO box 655474, Mail Station 238, Dallas, TX 75265, USA, ferguson@csc.ti.com, Tel: 214 995-0348, Fax: 214 995-0304.

Forbes, Harold C., Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, harold@cc.gatech.edu, Tel: 404 894-3982, Fax: 404 853-9378.

Fornaro, Robert J., NC State University, PO Box 8206, Raleigh, NC 27695, USA, fornaro@adm.csc.ncsu.edu, Tel: 919 737-7848, Fax: 919 737-3964.

Fujinami, Nobuhisa, Sony Computer Science Laboratory, Inc., Takanawa Muse Bldg., 3-14-13 Higashi-gotanda Shinagawa-ku, Tokyo 141 JAPAN, fnami@csl.sony.co.jp, Tel: +81-3-3448-4380, Fax: +81-3-3448-4273.

Gheith, Ahmed, IBM, 1400 Burnet Rd. #2812, Austin, TX 78758, USA, gheith@futserv.austin.ibm.com, Tel: 512 823-2406.

Ghosh, Kaushik, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, kaushik@cc.gatech.edu, Tel: 404 894-6169, Fax: 404 853-9378.

Gordon, Karen D., Institute for Defense Analyses, 1801 N. Beauregard St., Alexandria, VA 22311, USA, gordon@ida.org, Tel: 703 845-6630, Fax: 703 845-6848.

Gruensteidl, Guenter, Institut fuer Technische Informatik, Technical University of Vienna, Treitlstrasse 3 182 1, Vienna, Austria, A-1040, gruen@vmars.tuwien.ac.at, Tel: +43 222 58201 8170, Fax: +43 222 569149.

Halang, Wolfgang A., Groningen University, PO Box 800, 9700 Av. Groningen, The Netherlands, halang@cs.rug.nl, Fax: 001-31-50-633800.

Haritsa, Jayant R., Univ. of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706, USA, haritsa@cs.wisc.edu, Tel: 608 262-6625.

Heitmeyer, Connie, Naval Research Lab, Code 5534, Washington, DC 20375, USA, heitmeyer@itd.nrl.navy.mil, Tel: 202 767-3596.

Huizinga, Dorota M., Wayne State University, Detroit, MI 48165, USA, dmb@cs.wayne.edu (or is it dmh@cs.wayne.edu?), Tel: 313 486-0047.

Jahanian, Farnam, IBM Research, P.O. Box, Yorktown Heights, NY 10598, USA, farnam@watson.ibm.com, Tel: 914 784-7498.

Jameson, David H., IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA, dhj@rhun.watson.ibm.com.

Jeffay, Kevin, University of North Carolina, CB #3175 Sitterson Hall, Chapel Hill, NC 27514, USA, jeffay@cs.unc.edu.

Kamenoff, I. Nick, Fordham Univ., 336 John Mulcahy Hall, Bronx, NY 10458-5198, USA, bitnet: kamenoff@fordmurh, Tel: 212 579-2588, Fax: 212 579-2708.

Kiskis, Daniel L., Real-Time Computing Laboratory, The University of Michigan, EECS Building, 1301 Beal Ave., Ann Arbor, MI 48109-2122, USA, dlk@eecs.umich. edu, Tel: 313 763-6131.

Koob, Gary M., Computer Science Division, Office of Naval Research, Code 1133, 800 N. Quincy St., Arlington, VA 22217-5000, USA, koob@nrl-css.arpa, Tel: 202 696-0872.

Kopetz, Hermann, Technical University of Vienna, Treitlstrasse 3 182.1, A-1040 Vienna, Austria, hk@vmars.tuwien.ac.at, Tel: +43-1-58801 8180, Fax: +43-1-569149.

Labaw, Bruce, Naval Research Lab, Code 5534, Washington, DC 20375, USA, labaw@itd.nrl.navy.mil, Tel: 202 767-3249

Lee, Insup, University of Pennsylvania, Dept. of CIS, Philadelphia, PA 19151, USA, lee@central.cis.upenn.edu, Tel: 215 898-3532.

Lee, Juhnyoung, Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903, USA, jl2q@virginia.edu, Tel: 804 982-2296.

Lee, Yann-Hang, University of Florida, Computer & Information Sciences Dept. Gainesville, FL 32611, USA, yhlee@cis.ufl.edu, Tel: 904 392-1536, Fax: 904 392-1220.

Lehoczky, John P., Carnegie Mellon University, Department of Statistics, Pittsburgh, PA 15213-3890, USA, jpl@k.gp.cs.cmu.edu, Tel: 412 621-5473.

Lin, Kwei-Jay, Universit of Illinois, 1304 W. Springfield Ave., Urbana, IL 61801, USA, klin@cs.uiuc.edu, Tel: 217 333-1424.

Lintulanpi, Raino, VTT, Comp Tech Lab, PO Box 201, SF-90571 OULU; Finland, Tel: +358 18 509 111, Fax: +358 18 509 680.

Liu, Jane W. S., University of Illinois, 1304 W. Springfield Ave., Urbana, IL 61801, USA, janeliu@cs.uiuc.edu, Tel: 217 333-0135.

Locke, Doug, IBM Corp., 6600 Rockledge Dr., Bethesda, MD 20817, USA, cdl@cs.cmu.edu, Tel: 301 493-1496, Fax: 301 493-1746.

Markatos, Evangelos, Computer Science Dept., University of Rochester, Rochester, NY 14627, USA, markatos@cs.rochester.edu, Tel: 716 275-7230.

Marlowe, Thomas, Seton Hall University, 400 So. Orange Ave., South Orange NJ 07079, USA, marlowe@paul.rutgers.edu, Tel: 201 761-9784.

Maynard, David, Carnegie Mellon University, Department of ECE, Pittsburgh, PA 15213, USA, dpm@cs.cmu.edu, Tel: 412 268-7101.

Mok, Al, University of Texas at Austin, Dept. of Computer Science, Austin, TX 78712, USA, mok@cs.utexas.edu, Tel: 512 471-9542.

Natarajan, Swaminathan, Dept. of Computer Sciences, Texas A&M University, College Station, TX 77843-3112, USA, swami@cs.tamu.edu, Tel: 409 845-8287, Fax: 409 847-8578

Niehaus, Douglas, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, USA, niehaus@legato.cs.umass.edu, Tel: 413 545-4753, Fax: 413 545-1249.

Nirkhe, Vivek, Dept. of Computer Science, University of Maryland, A.V. Williams Bldg., College Park, MD 20742, USA, vivek@cs.umd.edu, Tel: 301 405-2724.

Oh, Henry Y., Dept. of Computer Science, University of Virginia, Thornton Hall, Charlottesville, VA 22903, USA, yosu@virginia.edu, Tel: 804 982-2291.

Paul, Chakkalamattam J., Carnegie Mellon University, ECE Department, Pittsburgh, PA 15213, USA, cjpaul@bahamas.ece.cmu.edu, Tel: 412 268 7121, Fax: 412 268-3890.

Pazy, Offer, Intermetrics, 733 Concord Ave., Cambridge, MA 02138, USA, offer@inmet.inmet.com, Tel: 617 661-1840.

Rajkumar, Ragunathan, IBM Research, P.O. Box 704, Yorktown Heights, NY 10598, USA, rajkumr@watson.ibm.com, Tel: 914 784-7931, Fax: 914 784-7455.

Ramamritham, Krithi, COINS, University of Massachusetts, Graduate Research Center, Amherst, MA 01003, USA, krithi@cs.umass.edu, Tel: 413 545-0196, Fax: 413 545-1249.

Rosenbaum, David, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, daver@cc.gatech.edu, Tel: 404 894-3982, Fax: 404 853-9378.

Schwan, Karsten, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, schwan@cc.gatech.edu, Tel: 404 894-2589, Fax: 404 853-9378.

Shin, Kang G., University of Michigan, Dept of EECS, 2225 Engineering Bldg. 1, Ann Arbor, MI 48109-2122, USA, kgshin@alps.eecs.umich.edu, Tel: 313 763-4617.

Smith, James G., Office of Naval Research, Code 12-Room 528, 800 North Quincy St., Arlington, VA 22217-5000.

Son, Sang H., University of Virginia, Dept. of Computer Science, Charlottesville, VA 22903, USA, son@cs.virginia.edu, Tel: 804 982-2205, Fax: 804 982-2214.

Stankovic, J., Dept. of Computer and Info. Science, University of Massachusetts, Amherst, MA 01003, USA, stankovic@cs.umass.edu, Tel: 413 545-0720, Fax: 413 545-0196.

Stewart, David B., Carnegie Mellon University, ECE Dept., Pittsburgh, PA 15213, USA, stewart@faraday.ece.cmu.edu, Tel: 412 268-7120, Fax: 412 268-3890.

Stone, Don, University of North Carolina, CB #3175 Sitterson Hall, Chapel Hill, NC 27514, USA, stone@cs.unc.edu, Tel: 919 962-1836.

Stoyenko, Alexander D., New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA, alex@vienna.njit.edu, Tel: 201 96-5765, Fax: 201 596-5777.

Strosnider, Jay K., Carnegie Mellon University, ECE Dept., Pittsburgh, PA 15213, USA, jks@usa.ece.cmu.edu, Tel: 412 268-6927, Fax: 412 268-3890.

Tokuda, Hideyuki, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3890, USA, hxt@cs.cmu.edu, Tel: 412 621-5473.

van Tilborg, Andre M., Office of Naval Research, Director, Computer Science Division, Code 1133, 800 North Quincy Street, Arlington, VA 22217-5000, USA, avantil@nswc-wo.arpa, Tel: 202 696-4302.

Vila, Juan, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA, jvila@cc.gatech.edu, Tel: 404 894-6169, Fax: 404 853-9378.

Wedde, Horst, Wayne State University, Detroit, MI 48202, USA, hwedde@zeus.cs.wayne.edu, Tel: 313 577-0731.

Zhao, Wei, Texas A&M University, Computer Science Department, College Station, TX 77843-3112, USA, zhao@cs.tamu.edu, Tel: 409 845-5098.

Zhou, Hongyi, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, USA, hongyi@cc.gatech.edu, Tel: 404 894-3982, Fax: 404 853-9378.

Final Report to IEEE

UNIVERSITY OF MASSACHUSETTS
AT AMHERST

Krithi Ramamritham, COINS Dept.
Lederle Graduate Research Center
Amherst, MA 01003
Tel: (413) 545-0196
FAX: (413) 545-1249
CSNet address: Krithi@NIRVAN.CS.UMASS.EDU

January 23, 1992

Ms. Anne Marie Kelly
Director of Conferences and Tutorials
IEEE Computer Society
1730 Massachusetts Ave., NW
Washington, DC 20036-1903

Dear Ms. Kelly:

Enclosed is our *final* TMRF for the 8th IEEE Workshop on Real-Time Operating Systems which was held May 15-17, 1991 in Atlanta.

Also included is a detailed analysis of the total expenditures for this workshop which includes the funds in the amount of $10,000 which were received from the Office of Naval Research.

The assistance afforded us by you and your staff was greatly appreciated.

Sincerely,

Krithi Ramamritham

Enclosures

IEEE COMPUTER SOCIETY
TECHNICAL MEETING REQUEST FORM
Request for Computer Society Approval of a Conference, Symposium or Workshop

1. ABOUT THIS FORM

This form is to be used to (1) request that the Computer Society (CS) "sponsor", "co-sponsor", or "cooperate in" a technical meeting (Conference, Symposium, Workshop), or (2) file part of the final report for an approved technical meeting. For the request, complete the "estimates", for the final report, complete the "actuals".

To request sponsorship or co-sponsorship, please complete all sections of the form. To request cooperation, please complete sections 1 through 10, and M11 and T9.

For all meetings for which support is being sought for the first time, or for which there is reason to believe that supplementary information would speed the Computer Society review process, a supplementary information sheet should be attached which addresses the following points if they are not addressed elsewhere on the form: 1) Sponsors: if there is reason to believe that the Computer Society might not know a sponsoring or cooperating entity, for example, if it was recently formed, it would be best to include a brief description of the sponsor, its charter, its founders, its membership, and indicate if it is or is not a for-profit organization; 2) Technical Program: indicate what steps will be taken to assure the quality of the technical program, what will the paper review process be, etc. 3) Registration fees: what will registration fee structure be, will IEEE Computer Society members and members of other sponsoring and cooperating organizations be eligible for lowest registration rates (except for student rates and other special discounted rates, such as for retired members). 4) Publicity: will the meeting be publicized in such a way that IEEE members will have the opportunity to become aware of it, 5) Computer Society members involvement: what will be the involvement of Computer Society members in the technical and administrative operation of the meeting, 6) Schedule: adequate time (at least 9-12 months) should be allowed between proposal submission and meeting dates, 7) Attachments: the Draft Call for Papers and other relevant information should be enclosed, 8) Proceeds: a clear statement should be made indicating to whom proceeds are to go.

Please follow the guidelines in the Computer Society Conference Handbook; they are keyed to this form. Table V provides "Rule-of-Thumb Costs" to help complete the form.

Since a number of copies will be made of the completed forms, please use a typewriter or felt pen to make the entries, which should be made in US dollars. For meetings held outside the USA, indicate here the local currency (e.g., Swiss Francs) and the conversion rate used.

Local Currency _____ Conversion Rate _____ Local currency units per US dollar

This form will be valid until the end of 1989; after that time, contact the Director of Conferences for a more recent one.

Send completed form to: **IEEE Computer Society, Director of Conferences, 1730 Massachusetts Avenue, N.W., Washington, D.C., 20036-1903.** (Phone 202-371-1013. Fax 202 728-9614)

CHECK ONE:
REQUEST FOR CS SPONSORSHIP _____ CO-SPONSORSHIP X*
COOPERATION X**

*Co-sponsored by: Office of Naval Research
**In Cooperation with IFAC and IFIP.

APPROVED

3/12/91
DATE

SIGNATURE

JUN 1990
RECEIVED
C.S.I.E.E.E.
CONFERENCE DEPT.

08/89

—PLEASE PRINT OR TYPE—

2. MEETING TITLE, DATES, LOCATION

Official Title of Meeting: Eighth IEEE Workshop on Real-Time erating Systems and Soft are

Acronym: 8th RTOS

Location (full address): Georgia Tech, Atlanta, Georgia

Housing Facilities (if different) _____

Dates: May 15, 16, 17, 1991

3. STATEMENT OF GENERAL CHAIR & FINANCE CHAIR

I have a copy of the Computer Society's "Conference Handbook" and I understand my responsibilities as outlined there.

This form including the budget has been prepared to the best of my ability and is complete and accurate. I understand that whenever it appears that the meeting may be in financial trouble, the Director of Conferences must be consulted.

I agree to provide the final report, to return the Computer Society advance loan, if any, to return the Computer Society share of the surplus funds, if any, and to close all accounts, all within four months after the meeting.

Further, I understand that all rights to this technical meeting are the property of and belong to the sponsoring entities.

General Chair Name Prof. Krithi Ramamritham IEEE/CS Member No. 0451609

Signature K. Ramamritham Date 6/5/90 Phones: Office 413 545-0196 Home 413 549-6101

Address Computer and Information Science Dept., Lederle Graduate Research Center, A305

University of Massachusetts, Amherst, MA 01003

Fax: 413 545-1249 Compmail/E-Mail Address: krithi@nirvan.cs.umass.edu

Finance Chair Name Victor Yodaiken IEEE/CS Member No. _____

Signature Victor Yodaiken Date 6/5/90 Phones: Office 413 545-4753 Home _____

Address Computer and Information Science Dept., Lederle Graduate Research Center, A305

University of Massachusetts, Amherst, MA 01003

Fax: 413 545-1249 Compmail/E-Mail Address: yodaiken@cs.umass.edu

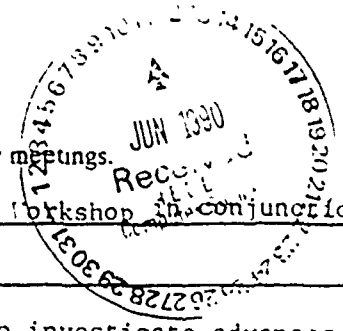
3A. STATEMENT OF TECHNICAL COMMITTEE CHAIR (if sponsored by Technical Committee)
I recommend approval of this meeting as submitted.

If the meeting has already been designated for a T/C Surplus Account, I understand sponsorship involves a financial commitment by the Technical Committee and that surpluses and losses will be apportioned as governed by current Computer Society policy.

T/C #1 Name Real-Time Systems T/C Chair Signature _____

T/C #2 Name _____ T/C Chair Signature Andre van Tilborg

T/C #3 Name _____ T/C Chair Signature _____



4. MEETING SCOPE, BENEFITS, ATTENDANCE

For first-time meetings, define scope and discuss overlap with approved Computer Society meetings.

This is the Eighth IEEE Real-Time Operating Systems and Software Workshop in conjunction with 17th IFAC/IFIP Workshop on Real-Time Programming

State benefits to society members: IEEE members will have an opportunity to investigate advances in real-time operating systems, software, and programming languages; to promote interaction among researchers and practitioners; to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

	ESTIMATED	ACTUAL
Attendance (from M11)	100	68
Sessions	10	10
No. of invited papers	0	0
No. of refereed papers submitted	55	63
No. of refereed papers accepted	20	24

5. SPONSORING & COOPERATING ENTITIES, & FINANCIAL COMMITMENT

List all entities, indicate if for-profit.

Entity	Representatives Name & Telephone	% Financial Commitment	Commitment Obtained Prelim.	Commitment Obtained Final
Computer Society:	TC Real-Time Systems	100%		
TC (if applicable)	Dr. Andre van Tilborg (202) 696-4312			
TC (if applicable)				
ACM:				
SIG (if applicable)				
	Office of Naval Research	$10,000		

6. SURPLUS & ADVANCE

	ESTIMATED	ACTUAL
Total income (from S1)	27750	$6,875.00
Total expenses (from S2)	20,625	16,875.00
Surplus (from S3)	2125	-0-

6. SURPLUS & ADVANCE (Continued)

Itemize expenditures requiring an advance:

Item	Amount
1. Administrative costs (postage, copy costs)	$ 1,500
2. Clerical Support	$ 500
3. Miscellaneous	$ 2,000
4.	$

Total Advance Loan requested from all cosponsors .... $ _____

Total Advance Loan requested from Computer Society .... $ _____

Partial Advance Loans requested from Computer Society and dates when needed:

$4,000.00 / 9 / 30 / 90 _____ / / / _____ / / / _____ / / /

7. ENCLOSURES

Enclose draft Call for Papers. If requesting sponsorship or co-sponsorship list below and enclose all contracts including Hotel and Exhibits, and any other material relating to financial obligations.

Enclosures	Yes	No
Call for Papers Enclosed	X	
Hotel Contract Enclosed		X
Other Contract Enclosed		

8. STEERING COMMITTEE MEMBERS

	Name	Employer	Phone-Office	Phone-Home
Chair				

9. TECHNICAL MEETING COMMITTEE MEMBERS

Please fill in this section completely with information on all committee members.

	Name	Employer	Phone-Office	Phone-Home
Co-General Chair #1	Krithi Ramamritham	Univ. of Massachusetts	413 545-0196	413 549-6101
General Co-Chair #2	Wolfgang A. Halang	Univ. of Groningen	31.50.633939	FAX 31.50.633976
Program Chair*	(as above)			
Finance Chair	Victor Yodaiken	Univ. of Massachusetts	413 545-4753	FAX 413 545-1249
Tutorials Chair*	n/a			

*Please attach page showing mailing address.

9. TECHNICAL MEETING COMMITTEE MEMBERS (Continued)

	Name	Employer	Phone-Office	Phone-Home
Exhibits Chair	---			
Publicity Chair	Krithi Ramamritham	U. Massachusetts	413 545-0196	413 549-6101
Registration Chair	Mrs. Betty Hardy	U. Massachusetts	413 545-4842	FAX 413 545-1249
Local Arrangement Chair	Karsten Schwan	GA Inst. of Tech.	404 894-2 589	
Publications Chair	---			
Audio-Visuals Chair	---			
Contact (Note: to be listed in Technical Meeting Schedule).	---			

10. PUBLICATIONS

	Yes	No
Proceedings	X**	
Published by CS	X**	
No. to be printed	100	
Sold by which Societies	**	
No. of pages	170	
Copyright assigned to IEEE	n/a	
If not, to whom		
Special issue of publication planned	X	
If yes, name of publication	Real-Time Systems Newsletter	
Approved by publication editor	X	

**Proceedings will be published after the workshop by Pergamon Press in the IFAC Proceedings Series.

11. MILESTONES

Meeting _____ Date _____

Where a range if given, longer time is for Conference, shorter for Workshops. Asterisked items generally not applicable to Workshops. Only items with "—" in front must have Name and Date entered to obtain CS approval.

BEFORE MEETING	Responsible Person(s)	Name	Date Due	Minimum Time Before Mtg.
—Define meeting, scope, etc.	General Co-Chairs	Krithi Ramamritham Wolfgang Halang	X	12-18 mos.
Appoint Committee	General Chair	Ramamritham/Halang	X	9-12 "
Sign Hotel Intent Letter	Local Arrangements Chair & Dir. of Conf.	Schwan/Ramamritham	8/31/90	10-14 "
Submit Proposal for Approval	General Chair	Ramamritham	X	9-12 "
—Hotel Contract Signed	Dir. of Conferences	Schwan/Ramamritham	8/31/90	9-12 "
Committee Meeting	General Chair	n/a		9-12 "
—Exhibit Sales Contract	Exhibit Chair & Dir. of Conferences	n/a		*-12 "
Open Bank Account	Finance Chair & Dir. of Conferences	Ramamritham/Yodaiken	9/30/90	8-11 "
—Call for Papers Prepared[1]	Program Chair	Ramamritham/Halang	X	8-11 "
—Place Call for Papers Magazine Ad[1]	Program Chair			8-11 "
—Finalize Publication Plans	Publication Chair	Ramamritham/Halang	12/1/90	*-11 "
—Papers/Summaries from Authors	Program Chair	Ramamritham	1/15/91	5-11 "
—Advance Announcement Prepared[1]	Publicity Chair			5- 6 "
—Advance Announcement Mag. Ad Prepared[1]	Publicity Chair			5- 6 "
Tutorial Speakers Contracts	Tutorial Chair & Dir. of Conferences			*- 6 "
Program Committee Meeting	Program Chair	(see note page A9#12)		4- 5 "
—Acceptance to Authors	Program Chair	Ramamritham/Halang	3/15/91	4- 5 "
—Author Kits to Authors	Publications Chair	n/a		*- 5 "
Plan Membership Booth	Local Arrang. Chair & Dir. of Press			*- 5 "
—Exhibit Sales Completed	Exhibits Chair	n/a		*- 5 "
Committee Meeting	General Chair	(see note page A9#12)		5- 5 "
—Advance Program Prepared[1]	Publicity Chair	Ramamritham	4/15/91	3- 4 "
—Press Release	Publicity Chair	n/a		3- 4 "
—Place Advance Program Magazine Ad[1]	Publicity Chair	n/a		3- 4 "
—Final Papers from Authors	Publicity Chair	Ramamritham	4/1/91	*- 3 "

Footnotes:
(1) All magazine ad copy 6 weeks prior to month of desired issue. All advertising pieces takes 7-8 weeks effort: Typesetting 5-7 days; Proofing 1-3 days; Printing 5-7 days; Mailing Lists 3 weeks; Mailing 3-5 days; Postal Service 3 weeks for third class mail.

11. MILESTONES (Continued)

BEFORE MEETING	Responsible Person(s)	Name	Date Due	Minimum Time Before Mtg.
Audio-Visuals Quality Reviewed	Audio-Visuals Chair			6- 3 wks.
—Proceedings to Printer	Publicity Chair	n/a		•-10 "
Final Session Room Assignments	Local Arrangements Chair	Ramamritham/Schwan	4/15/91	•- 8 "
Tutorial Notes to Director of Conferences	Tutorial Chair	N/A		•- 6wks.
Session Signs Ordered	Local Arrangements Chair	N/A		•- 5 "
—Final Program[1]	Publicity Chair	Ramamritham/Halang	4/5/91	•- 5 "
—Advance Registration Closes	Registration Chair	Hardy	4/15/91	+ 4 "
—Final Program Delivered	Registration Chair	Hardy	5/10/91	•- 3 "
Badges/Ribbons Ordered	Registration Chair	Schwan	5/10/91	3- 3 "
Hotel Food Quantity Estimated	Local Arrangements Chair	Schwan/Ramamritham	6/30/90	2- 2 "
—Proceedings Delivered to site	Publications Chair	Hardy	5/8/91	•- 3days
Hotel Food Quantity Guaranteed	Local Arrangements Chair	Schwan/Ramamritham	4/15/91	48 hrs. before event

AFTER MEETING				
Committee Debriefing	General Chair			Day after
Recommend Committee Awards	General Chair			4- 4 wks.
—Submit Interim Report & Return Advance	Gen. & Finance Chairs			2- 2 mos.
—Submit Final Report & Monies	Gen. & Finance Chairs	Ramamritham	9/1/91	4- 4 mos.
—Send Meeting Attendee List	General & Registration Chair	Ramamritham	6/30/91	4- 4 mos.

Footnotes:
(1) All magazine ad copy required 6 weeks prior to month of desired issue. All advertising copy takes 7-8 weeks effort. Typesetting 5-7 days; Proofing 1-3 days. Printing 5-7 days; Mailing Lists 3 weeks; Mailing 3-5 days; Postal Service 3 weeks for third class mail.

X=already completed
n/a= Not applicable

12. BUDGET

MEETING EXPENSES (Exclude tutorials and exhibits)	Estimate	Actual
M1 Advertising (including printing, handling, mailing)		
ATTACHED ADVERTISING WORKSHEET MUST BE COMPLETED.		
(a) Call-for-Papers (019)	$ 135.30	$ 145.00
(b) Announcement (021)	$ 135.30	$ 145.00
(c) Advance Program (020)	$ 195.30	$ 155.93
(d) Posters (022)	$ -0-	$
(e) Other (specify) (017)	$	$
	$	$
Subtotal for advertising	$ 465.90	$ 445.93
(e) Tutorial expense (if tutorial advertising is not budgeted separately, subtract 20% and add it to T1)	$ n/a	$
M1 Total Advertising	$ 465.90	$ 445.93
Advertising cost as % of total technical meeting cost (M1 / M10)	$ 2.25%	$ 2.64%
M2 Committee Expenses		
(a) Secretary (851) ..... No. hours ____ X $/hr. ____ =	$	$
(b) Telephone (830)	$ 50	$ 12.70
(c) Postage (570)	$ 115	$
(d) Committee Travel (871)	$ 1,000	$
(e) Reproduction (190)	$ 115	$
(f) Compmail [2] (830) ....and computer services	$ 100	$ 792.00
(g) Other (specify) (517)	$	$
	$	$
M2 Total Committee Expenses	$ 1,380	$ 804.70

Footnotes
(2) Technical meetings with budgets over $30K should include funds for a Compmail ☎ account for key members of the Steering and Tech. Meeting Committees.

A-8

12. BUDGET (Continued)

REMARKS It is our intent to conduct all "meetings" for this workshop via electronic mail; therefore, we will not be holding a committee meeting during the planning stages.

M3 Operating Expenses	Estimate	Actual
(a) Advance Registration		
(1) By Computer Society (761) (70% est. attendance X C_1 from Table IV)	$	$
(2) or by other means (show computation) (768) 30 hrs x $17.34 mail/telephone/registration	$ 520	$ -0-
(b) On-site registration[3] (762) one person 1/2 day 4 x $19.70	$ 78	$ 250.00
(c) Badges, tickets, evaluation forms (761)	$	$
(d) Security (771)	$	$
(e) Gratuities, awards, attendee travel[1] (030)	$	$
(f) Keynote and special addresses[1] (875)	$	$
(g) Audio Visuals and Microphones—Labor & Equipment (719)	$ 70	$ 130.25
(h) Typewriters and other equipment (713)	$	$
(i) Final program (artwork and printing) (600)	$	$
(j) Proceedings for attendees[4] (615)	$	$
No. copies 100 x $/page .054 x 170 No. pages.	$ 918	$ 590.21
Freight (615)	$ 100	$ -0-
(k) Signs (meeting rooms, other) (795)	$	$
(l) Shipping to Meeting ( )	$	$
(m) Meeting space rental (715)	$ 300	$ -0-
M3 Total Meeting Operating Expenses	$ 1,986	$ 970.46

Footnotes
(1) These items must be explained under Remarks.
(3) If done by CS Staff, enter C_2 from Table IV per staff member plus travel expenses.
(4) Cost per page if done by CS Press is given on Table III.

12. BUDGET (Continued)

M4 Other Technical Meeting Expenses

Include and describe any expense not identified on previous pages.

(a) Bank charges; Credit card service charges (070)	$	$
(b) Rebates (670)	$	$
(c) Bad debts (650)	$	$
(d) Insurance (396)	$	$
(e) Audit ( )	$	$
(f) Other (511) Administrative costs - badges, attendee list, reports, etc., 20 hrs. x $17.34 =	$ 260	$ 187.05
M4 Total Other Meeting Expenses	$ 260	$ 187.05

M5 Meeting Expenses Subtotal
Add M1, M2, M3, M4 $ 4092 $ 2,408.14

M6 Contingency (180)
Enter 5 to 15% of line M5 ($1000 minimum) ____ % $ 1,000 $ -0-

M7 Computer Society Administrative Services[5] (780)
Enter 14% of line M5 $ 573 $ 337.15

Footnotes:
(5) This is a mandatory entry for all meetings: it helps recover expenses incurred by the Computer Society for all technical meetings. For co-sponsored meetings, this expense will be remitted to the sponsors in proportion to their financial commitment as provided in Section 5, page A-3.

REMARKS

12. BUDGET (Continued)

M8 Social Functions

(a) Coffee, pastries, etc., between sessions (472)		
No. Breaks 5 X No. people 100 X $/person $6.00	$ 3,000	$ 2,082.73
(b) Luncheons (471)		
No. Luncheons 2 X No. people 100 X $/person 15.00	$ 3,000	$ 2,378.88
(c) Receptions (473)		
No. Receptions 1 X No. people 100 X $/person 25.00	$ 2,500	$ 3,491.14
(d) Banquets (474)		
No. Banquets ____ X No. people ____ X $/person ____	$	$
(e) Speakers Hospitality (475)		
No. people 20 X $/person 11.50 x 2 days	$ 460	$ 576.96
(f) Transportation (courtesy bus, etc.) (861)	$	$
(g) Other social function expenses (specify) (476)	$	$

NOTE: M8 above includes gratuities, taxes, and any related service fees.

M8 Total Social Function Expenses	$ 8,960	$ 8,529.71
Social cost per attendee	$ 90	$ 125.43

M9 Services from Computer Society Staff N/A

Use Table V to find the charge for any service desired and enter the amount below.

SERVICE	YES	CHARGE
Call for Papers Artwork/Typesetting Coordination	____	Include in M1 & Worksheet
Announcement Artwork/Typesetting Coordination	____	Include in M1 & Worksheet
Advance Program Artwork/Typesetting Coordination	____	Include in M1 & Worksheet
Final Program Artwork/Typesetting Coordination	____	Include in M3 & Worksheet

12. BUDGET (Continued)

M9 Services from Computer Society Staff (Continued)

SERVICE	YES	CHARGE Estimated	CHARGE Actual
Print & Mail Call for Papers	____	Include in M1	
Print & Mail Announcement	____	Include in M1	
Print & Mail Advance Program	____	Include in M1	
Print Final Program	____	Include in M3	
Prepare Budget (763)	____	$_____	$_____
Treasurer's Service (764)	____	$_____	$_____
Advance Registration[6]	____	Include in M3	
On-Site Registration	____	Include in M3	
Hotel Negotiations (765)	____	$_____	$_____
Other Negotiations (specify) (766)	____	$_____	$_____
_____			
Prepare & Place Press Releases	____	Include in M1 & Worksheet	
Prepare Advertisements	____	Include in M1 & Worksheet	
On-site Publications Sales	____	No charge	
On-site Membership Booth	____	No charge	
Proceedings Publication	X	Include in M3	
Other services of the Headquarters or Publication offices (specify) (767)			
Real-Time Systems Newsletter	X	$ 6,000*	$ 5,600.00
*this is based on $5,200 for printing and $800 for mailing. We anticipate approximately 150 pages.		$_____	$_____
_____		$_____	$_____
M9 TOTAL		$ 6,000	$ 5,600.00
M10 Total Technical Meeting Expenses Add M5, M6, M7, M8, M9		$ 20625	$ 16,875.00

Footnotes:
(6) The Director of Conferences should be treasurer for all conferences requesting advance registration processing by the Computer Society. A Finance Chair should still be appointed to the conference committee in this case.

MEETING INCOME
(Exclude tutorials & exhibits)

M11 Registration	Estimated	Actual
Advance Registration		
Members[7]	60 @ $130 = $7,800	19 @ $130 = $2,470
Non-members[8]	15 @ $155 = $2,325	5 @ $155 = $775
Full-time student members[9]	10 @ $55 = $550	15 @ $55 = $825
Other[10] (specify below)	___ @ $___ = $___	___ @ $___ = $___
Late / On-site Registration[11]		
Members[7]	10 @ $150 = $1,150	16 @ $150 = $2,400
Non-members[8]	5 @ $185 = $925	1 @ $185 = $185
Full-time student members[9]	0 @ $55 = $-0-	4 @ $55 = $220
Complimentary[10]	*8	
Other[10] (specify below)	___ @ $___ = $___	___ @ $___ = $___
Total attendance[12]	100	68
Last year's paid meeting attendance	___	

M11 Total Registration Income	$12,750	$6,875
M12 Other Income (specify)		
Publication sales (300)	$0	$___
Interest (140)	$0	$___
Other (706) Grant from Office of Naval Research	$10,000	$___
M13 Total Income (M11 plus M12)	$22,750	$6,875

Remarks 8 complimentary: 4 students from Georgia Tech were conference workers; Drs. Koob and Van Tilborg (ONR); and Krithi Ramamritham and Karsten Schwan.

Footnotes
(7) Members of the Computer Society, IEEE, co-sponsoring or cooperating entities
(8) Non-member rates should be 25-50% higher than member rate.
(9) Student rates usually are for sessions only and do not include Proceedings or social functions; if otherwise, please indicate under Remarks.
(10) Specify, under Remarks, who will receive complimentary or special rates, and indicate if the rate includes a copy of the Proceedings and attendance at social functions. If committee members, speakers, session chairs, etc., will receive complimentary or special rates, they must be listed here or they must pay the appropriate member or non-member rate. Use discretion. Retired members are entitled to reduced rates. Special combination rates offering discounts for attending two functions must be shown ie. Conf. + 1 Tutorial, Conf. + 2 Tutorials or 2 Tutorials etc.
(11) Late/On-site Registration rates should be at least 20% higher than advance.
(12) An estimate more than 10% higher than last year's actual should be explained.

TUTORIAL EXPENSES N/A

	Estimated	Actual
T1 Advertising (including printing, handling, mailing). Complete attached worksheet. If tutorial advertising is not budgeted separately, enter amount from M1.e		
T1 Total Tutorial Advertising (026)	$_____	$_____
T2 Operating Expenses		
(a) On-site registration[13] (768)	$_____	$_____
(b) Security (772)	$_____	$_____
(c) Gratuities (031)	$_____	$_____
(d) Audio Visuals and Microphones—Labor & Equipment (717)	$_____	$_____
(e) Typewriters and other equipment (720)	$_____	$_____
(f) Texts (613) No. copies _____ X $/copy_____	$_____	$_____
(g) Notes (514) No. copies _____ X $/copy_____	$_____	$_____
(h) Signs (798)	$_____	$_____
(i) Speaker fees and travel expenses		
No. of tutorials _____ No. of days _____		
No. of full-day speakers _____ x Rate[14]_____ (637)	$_____	$_____
No. of half-day speakers _____ x Rate[14]_____ (637)	$_____	$_____
No. of speakers _____ X travel expense/speaker _____ (872)	$_____	$_____
(j) Meeting space rental (721)	$_____	$_____
T2 Total Tutorial Operating Expenses	$_____	$_____
T3 Other Tutorial Expenses (512) Include and describe any expense not identified above		
_____	$_____	$_____
_____	$_____	$_____
_____	$_____	$_____
T3 Total Other Tutorial Expenses[4]	$_____	$_____

Footnotes
(13) If this will be done by Computer Society staff, enter C_2 from Table IV plus travel expenses.
(14) Refer to Table V.

EXHIBIT EXPENSES

	Estimated	Actual
E1 Advertising (including printing, handling, mailing) (015)		
Complete attached advertising worksheet. If exhibit advertising is not budgeted separately, enter a pro-rated amount of the advertising budget for the meeting.		
E1 Total Exhibit Advertising	$_____	$_____
E2 Operating Expenses		
(a) Registration Services	$_____	$_____
(b) Space Rental (714)	$_____	$_____
(c) Management Fee (635)	$_____	$_____
(d) Security (773)	$_____	$_____
(e) Insurance (397)	$_____	$_____
(f) Busing (862)	$_____	$_____
(g) Drayage (712)	$_____	$_____
(h) Other expenses (specify) (516)	$_____	$_____
_____	$_____	$_____
_____	$_____	$_____
E2 Total Exhibit Operating Expenses	$_____	$_____
E3 Exhibit Expenses Subtotal Add E1, E2	$_____	$_____
E4 Contingency Enter 5 to 15% of line E3 _____% (182)	$_____	$_____
E5 Computer Society Administrative Services[5] (782) Enter 14% of line E3	$_____	$_____
E6 Total Exhibit Expenses Add E2, E3, E4, E5	$_____	$_____

Footnote:
(5) This is a mandatory entry for all meetings; it helps recover expenses incurred by the Computer Society for all technical meetings. For co-sponsored meetings, this expense will be remitted to the sponsors in proportion to their financial commitment as provided in Section 5, page A-3.

EXHIBIT EXPENSES (Continued)

	Estimated	Actual
EXHIBIT INCOME		
E7 Exhibitor Fee Income		
No. Exhibitors _____ X $/Exhibitor _____ (410) =		
or No. Booths _____ X $/Booth _____ (410) =	$_____	$_____
E8 Other Exhibit Income (specify) (708)	$_____	$_____
_____	$_____	$_____
_____	$_____	$_____
E9 Total Exhibit Income (E7 plus E8)	$_____	$_____

REMARKS
1. Describe exhibit facilities, state cost per sq. ft. sales price of booth space, attach copy of any contracts, etc.

_____

_____

_____

S BUDGET SUMMARY	Estimated	Actual
S1 INCOME		
M13 Total Meeting Income	$ 22,750	$ 6,875
T11 Total Tutorial Income	$ -0-	$
E9 Total Exhibit Income	$ -0-	$
S1 TOTAL INCOME	$ 22,750	$ 6,875
S2 EXPENSES		
M10 Total Meeting Expense	$ 20625	$ 16,875
T8 Total Tutorial Expense	$ -0-	$
E6 Total Exhibit Expense	$ -0-	$
S2 TOTAL EXPENSES	$ 20625	$ 16,875
S3 SURPLUS[15] (S1 minus S2)	$ 2125	$ [10,000]

Footnotes:
(15) Estimated surplus should be at least 10% of estimated expenses.

DRAFT

EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE
(in conjunction with)
17th IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING

CALL FOR PAPERS

May 15 - 17, 1991
Atlanta, GA, USA

This workshop has several goals:

- to investigate advances in *real-time* operating systems, software, and programming languages,
- to promote interaction among researchers and practitioners,
- to evaluate the maturity and evolutionary directions of real-time programming theories and approaches.

Workshop attendees will explore current ideas on real-time software, programming languages, and operating systems. Position papers describing new ideas, promising approaches, and work in progress are considered particularly appropriate.

Possible topics of this workshop include:

- Real-time operating systems,
- Real-time programming, requirements analysis and specification,
- Evaluation of real-time systems,
- Real-time scheduling and resource management,
- Examples of real-time (control) systems with challenging time constraints.

Prospective attendees should send 10 copies of a (no more than) 5-page position paper *Krithi Ramamritham* by January 15, 1991. To facilitate the reviewing process, it is recommended that submission also be sent electronically—in the form of plain ASCII files. The position paper should focus on insights and lessons gained from recent researc and practical experience in real-time operating systems and software. Complete details regarding the workshop will be sent to all participants along with acceptance letters by March 15, 1991. Preprints of the accepted papers will be made available at the Workshop. Proceedings will be published after the workshop by Pergamon Press in the IFAC Proceedings Series. Attendance will be limited to approximately 75 active worker in the field.

WORKSHOP CHAIRS

Krithi Ramamritham
Dept. of Computer & Info. Science
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003
USA
413 545-0196
krithi@nirvan.cs.umass.edu

Wolfgang A. Halang
Dept. of Computing Science
University of Groningen
P. O. Box 800
NL-9700 AV Groningen
The Netherlands
+31-50-63 39 39
halang@cs.rug.nl

PROGRAM COMMITTEE

- Robert C. Cook, University of Virginia
- Juan A. de la Puente, Polytech Univ. of Madrid
- Wolfgang D. Ehrenberger, Soc. of Reactor Safety, Munich
- Farnam Jehanian, IBM Yorktown Heights
- Hermann Kopetz, Tech University Vienna
- Michael G. Rodd, University Wales, Swansea
- Karsten Schwan, Georgia Tech
- Janos Szlanko, Central Res. Inst. for Physics, Budapest
- Hide Tokuda, Carnegie-Mellon Univ.,PA
- T. J. Williams, Purdue University
- Wei Zhao, U. of Adelaide, SA

Co-sponsored by:
- IEEE Computer Society
  Technical Committee on Real-Time Systems
- Office of Naval Research

In cooperation with:
- IFAC
  Technical Committee on Computers
  Working Group on Real-Time Programming

and
- IFIP Working Group 5.4
  on Computerized Process Control

IEEE

Re: Page A-4, Item #9

Prof. Krithi Ramamritham
Computer and Information Science Dept.
University of Massachusetts
Lederle Graduate Research Center, Room A309
Amherst, MA 01003

Prof. Wolfgang A. Halang
Dept. of Mathematics and Computing Science
University of Groningen
PO Box 800
NL-9700 AV Groningen
The Netherlands

IFAC/IFIP Report

July 31, 1991

Prof. Juan A. de la Puente
Grupo de Ingeneria de Control
ETSI Telecomunicacion
Ciudad Universitaria
E-28040 Madrid
SPANJE

Dear Juan,

Enclosed please find the Final Report on the **17th IFAC/IFIP Workshop on Real Time Programming**. I should highly appreciate if you could take this report along to the forthcoming meeting of the IFAC Technical Committee on Computers and hand it over to the chairman. A copy of the report has already been sent to the IFAC Secretariat.

Thank you very much in advance and best regards.

Yours sincerely,

Prof. Dr. Wolfgang A. Halang

encl: Final Report

cc: IFAC Secretariat Laxenburg

Joint
17th IFAC/IFIP Workshop on Real Time Programming
and
Eight IEEE Workshop on Real-Time Operating Systems and Software

Final Report

15 – 17 May 1991
Atlanta, GA, U.S.A.

1. Breakdown of Attendance by Country

Austria	2
Finland	1
Great Britain	1
Japan	1
The Netherlands	1
Spain	1
U.S.A.	61

2. Method and Statistics of Paper Selection

Out of 63 submissions to the Workshop 24 papers were selected for presentation. The contributions came from Europe, North America, and South America: 46 from academia, 16 from industry, and one from a government agency. Of the 63 papers submitted

47 were from U.S.A.	37 from universities
	9 from industry
	1 from government
13 from Europe	7 from universities
	6 from industry
3 from elsewhere	2 from universities
	1 from industry

Of the 24 accepted papers

21 were from U.S.A.	16 from universities
	4 from industry

	1 from government
2 from Europe	1 from university
	1 from industry
1 from elsewhere	1 from university

Each paper was refereed by four reviewers, who gave marks. The highest scoring papers were selected for acceptance and assigned to presentation in six sessions.

3. Brief Summary of Programme and Discussion

The event's primary focus was on software development for real time systems and on real time operating systems. The six sessions addressed the subject areas scheduling, operating systems, design and tools, programming languages, timing analysis, and experience and case studies.

In addition to the discussions that took place in each of these sessions and during the breaks, the Workshop devoted ample time for focussed discussions by arranging four panels addressing the following topics: fault tolerance, programming languages, scheduling, and operating systems. After short statements by the panelists lively discussions ensued with a large number of contributions from the audience. The Proceedings contain reports summarising the opinions expressed during the panel discussions.

4. Comments on Translation Arrangements

The Workshop was held in English and no translation into other languages was provided.

5. Budget and Actual Expenses

In the following financial record the abbreviation ONR stands for Office of Naval Research, an agency of the United States Navy which co-sponsored the Workshop.

INCOME:			
Registration fees:			
19	members	@$130	2470
16	late-pay members	@$150	2400
35	Total Members		
5	non-members	@$155	775
1	late-pay non member	@$185	185
6	non-members		
19	students	@ $55	1045
60	Total Paid Participants		
4	Complimentary		
4	Conference Workers		

68 Total Attended	
Total Registration Income	$6,875
Grant Office of Naval Research	$10,000
TOTAL INCOME	$16,875

MEETING EXPENSES:	IEEE	ONR
Meeting Functions:		
Conference Preprints and Programme Printing, copying, binding, mailing, fax, courier service	777.26	514.03
Total Promotion:	$1291.29	
Meeting Facilities:		
Meeting Facilities	Complimentary	
Audio-visual equipment	130.25	
Total Meeting Functions:	$130.25	
Administrative Costs:		
Clerical @$17.34/hour		600.00
On-site registration	250.00	
Computer costs		100.00
Total Administrative Costs:	$950.00	

SOCIAL FUNCTION EXPENSES:		
Reception:	3416	
Speakers' Breakfasts:	576	
Luncheons:	2379	
Breaks:	2083	
Birds of Feather Session	165	
Total Food and Function:	8455 ($124 per person	
Exact Total Social Functions:	$8453.51	
IEEE share	$5717.78	
ONR share	$2737.22	

GRAND TOTALS:	IEEE	ONR	Total
EXPENSE:	$6875.29	$3951.25	$10826.54
INCOME:	$6,875	$10,000	$16,875

6. Comments on New Features Tested

Since the participation in the Workshop from North America was unsatisfactory in recent years, the Working Group on Real Time Programming decided in 1988 to hold the meeting in the United States in 1991. To avoid competition and fragmentation, the Workshop was organised jointly with a thematically related national event, viz., the IEEE Workshop on Real-Time Software and Operating Systems. As a member of the American Automatic Control Council, IFAC's National Member Organisation in the U.S.A., the IEEE sponsored this joint Workshop and took the financial responsibility. As the above-mentioned attendance figures show, this kind of collaboration with other organisations having a similar scope as the IFAC WGRTP seems to open new interesting possibilities for future events.

7. Comments on IFAC/Automatica Publicity

The Workshop was publically announced in the IFAC Newsletter, the Informatik-Spektrum of (the German) Gesellschaft für Informatik, the IEEE Computer Magazine, the IEEE Real-Time Systems Newsletter, on electronic bulletin boards, and by mailing electronic or hard-copy versions of the call-for-papers to several hundred addresses.

8. Publication

Preprints were prepared and handed out at the Workshop containing the accepted position papers. The full-length papers are published in the official proccedings appearing in the IFAC Symposia Series. The deadline for submitting the camera-ready versions to Mr. Strange in Oxford was 31 July 1991. No paper was recommended for publication in *Automatica*.

Wolfgang A. Halang
University of Groningen
Chairman, IPC

Krithi Ramamritham
University of Massachusetts
Chairman, NOC

Appendix A

APPLICATION FOR IFAC SPONSORED MEETING

1. National Member Organization accepting full financial responsibility
   Name: IEEE (in cooperation with AACC)
   Mailing Adress:
   Tel./Telex:
2. The proposed meeting is to be a: Symposium ☐ Conference ☐ Workshop ☒ (check one)
3. Proposed meeting title: Joint 8th IEEE Workshop on Real-Time Software and Operating Systems and 17th IFAC/IFIP Workshop on Real-Time Programming
4. Brief statement of meeting scope:
   Engineering aspects of software for real-time systems, esp. computer control systems. Particular areas of interest include specification and design methods for real-time systems, languages for real-time programming, real-time operating systems, real-time data bases and programming environments and tools for real-time systems.
5. Suggested IFAC Technical Committee(s) Sponsor: TC on Computers
6. Suggested other international organizations co-sponsoring with IFAC: IEEE Computer society
7. Location and date of meeting: Atlanta, Georgia May 15, 16, and 17, 1990
8. Relationship of meeting to other events known to the applicant such as other IFAC activities, other non-IFAC activities, exhibits: none
9. Expected attendance: 80 - 100
10. Approximate registration fee (including preprints) in Swiss Francs: 225
11. Conference language(s)? English
    Will simultaneous translation be used during the meeting? no
12. Will proceedings be published in other language(s) in addition to English? no
    If so what language(s)?
13. Chairman
    National Organizing Committee
    Name: Krithi Ramamritham
    Title/Position: Associate Professor, Computer and Information Science Dept
    Mailing Address: University of Massachusetts
    Lederle Graduate Research Center, Amherst, MA 01003
    Tel./Telex: tel. 413 545-0196 , fax 413 545-1249
14. Chairman
    International Program Committee
    Name: Krithi Ramamritham
    Title/Position: Associate Professor, Computer and Information Science Dept
    Mailing Address: University of Massachusetts
    Lederle Graduate Research Center, Amherst, MA 01003
    Tel./Telex: tel. 413 545-0196 fax 413 545-1249
    AND
15. Symposium Editor
    Name: Wolfgang A. Halang
    Title/Position: Professor
    Mailing Address: University of Groningen, Dept. of Computing Science
    P. O. Box 800, 9700 AV Groningen, The Netherlands
    Tel./Telex: Tel. +31-50-633925, Fax +31-50-633976, Tlx. 53410 rugro
16. Person submitting this application
    Name: W. A. Halang and K. Ramamritham
    Title/Position:
    Mailing Address: same as #14 and 15
    Tel./Telex:

I agree to abide by the IFAC Regulation laid down in the appropriate Guideline including the requirement that the IFAC Publications Managing Board will determine whether or not the Proceedings are to be published by Pergamon Press, the official IFAC publisher.

Signature of submitter: W. A. Halang K. Ramamritham Submission Date: 10-15-'90

Budget Detail

EIGHTH IEEE WORKSHOP ON REAL-TIME OPERATING SYSTEMS AND SOFTWARE
(in conjunction with)
17th IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING

FINAL FINANCIAL REPORT

INCOME:				
Registration fees:		IEEE	ONR	Total
19 members	@$130	2470		
16 late-pay members	@$150	2400		
35 Total Members				
5 non-members	@$155	775		
1 late-pay non member	@$185	185		
6 non-members				
19 students	@ $55	1045		
60 Total Paid Participants				
4 Complimentary				
4 Conference Workers				
68 Total Attended				
Total Reigstration Income		$68?5		
Grant/Office of Naval Research			$10000	
TOTAL INCOME				$16,875

MEETING EXPENSES:		IEEE	ONR	TOTAL
Meeting Functions:				
Conference Proceedings and Program		777.26	458.63	
Printing, copying, binding				
Mailing, fax, federal expr...				
Total Promotion:				$1235.89
Meeting Facilities:				
Meeting Facilities		Complimentary		
Audio-visual equipment		130.25		
Parking		Complimentary		
Total Meeting Functions:				$130.25
Administrative Costs:				
Registration/mail/t...				
Fiscal Administrati...				
Submissions processing				
On-site registration		370.00		
Computer costs			792.00	
Total Administrative Costs				$1042.00
Total Meeting Expenses		$1157.51	$1250.63	$2408.14
SOCIAL FUNCTION EXPENSES:				
Reception:	$491			
Speakers' Breakfasts:	298			
Luncheons:	2493			
Breaks:	2083			
Birds of Feather Session	165			
Total Food and Function	8530	5717.49	$2812.22	$ 8529.71
$125 per person				
TOTAL EXPENSE		$6875.00	$4062.85	$10937.85
14% of Total meeting Expenses to IEEE			337.15	
PO for Netwletter			5600.00	
Total Expense		$6875.00	$10000.00	$16875.00
TOTAL INCOME:		$6875.00	$10000.00	$16875.00

DEPARTMENT OF THE NAVY
OFFICE OF NAVAL RESEARCH, CODE 1513:GDB
800 NORTH QUINCY STREET

ARLINGTON, VIRGINIA 22217-5000

GRANT NO: N00014-91-J-1369

R&T PROJECT: 4331791---01
ACO CODE: N68883
CAGE CODE: 4B831
DISBURSING CODE: N00179

SYMPOSIUM GRANT

GRANTEE: University of Massachusetts
Munson Hall
Amherst, MA 01003

DUPLICATE ORIGINAL

APPROPRIATION: AA 1711319.W1AE

Object Class:	000
Unit Ident Code:	RA434
Suballctment:	0
Auth. Acct. No:	068342
Transaction Type:	2B
Prop. Acct. Act.:	000000
Cost Code:	015080000100
Amount:	$10,000.00
FRC:	4331

R&T Project Code: 4331791---01, Dated: 08 AUG 1990

TOTAL GRANT AMOUNT: $10,000.00

AUTHORITY: 10 USC 2358 as amended, and 31 USC 6304.

GRANT PURPOSE: The Purpose of this Grant is to provide partial funding to support the Eighth IEEE Workshop in Real-Time Operating Systems.

The conduct of the workshop, the personnel and effort and the use of funds for direct and indirect expenses shall generally be as set forth in the Grantee's proposal entitled "Eighth IEEE Workshop on Real-Time Operating Systems and Software", dated 29 JUN 1990 which proposal is incorporated herein by reference. The Grantee agrees to obtain concurrence of the Grantor for any desired deviation from the proposal.

PERIOD: The Grant is for the period 01 NOV 1990 through 30 NOV 1991.

PRINCIPAL INVESTIGATOR: The Principal Investigator, Professor Krithi Ramamritham shall be continuously responsible for the conduct of the project. The Grantee agrees to obtain approval of the Grantor before changing the Principal Investigator.

SCIENTIFIC OFFICER: The Scientific Officer representing the United States Government under this Grant is Andre M. Van Tilborg, Code 1133, Office of Naval Research, 800 North Quincy Street, Arlington, Virginia 22217-5000.

GRANTS ADMINISTRATOR: The Grants Administrator for this Grant is:
Office of Naval Research
Resident Representative N68883
Charles S. Draper Laboratory

555 Technology Square MS54
Cambridge, MA 02139-3539

PAYMENTS: Upon submission of invoices by the Grantee in accordance with the provisions of this Grant, the amount specified herein shall be paid as follows: $10,000.00 payable on 01 NOV 1990. Invoices hereunder shall be submitted by the Grantee in sextuplicate to the Grants Administrator for certification and transmittal to the Navy Regional Finance Center, Crystal Mall #3, Rm. 260 Attn: Code 431, Washington, D.C. 20371-5400, where payment will be made.

The Grantee is participating in the cost of this effort.

PERFORMANCE REPORTS AND/OR PROCEEDINGS: (a) The Grantee shall submit to the attached distribution list the following documents within 60 days after the end date of this grant:

5 copies of the Proceedings.

(b) The Grantee shall include a complete "Document Control Data - R&D" form (DD Form 1473) as the last page of each copy of every scientific and technical report prepared under this Grant. The form contains instructions for preparation. The cognizant Government Grant Administrator will provide assistance to the Grantee in obtaining the required forms. Administrative type reports, managerial (status) reports, and reprints submitted as technical reports are excluded from this requirement.

PATENTS AND COPYRIGHTS: (a) With respect to patents and other rights arising out of inventions, improvements or discoveries conceived or first actually reduced to practice during the effort, Grantee shall (1) give and hereby does grant to the United States Government an irrevocable, non-exclusive, non-transferable royalty-free license to practice or have practiced for its benefit, each invention (whether or not patentable) throughout the world, (2) advise Grantor of the filing of each patent application in any country and furnish a copy thereof to Grantor, (3) give Grantor the right to file patent application(s) for any invention on which Grantee does not intend to file, as to which inventions the Government is hereby granted sole and exclusive title, and (4) on request, furnish grantor duly executed instruments fully confirmatory of said license and/or title rights.

(b) The Government shall have the right to publish, translate, reproduce, deliver, and dispose of all data, including reports, drawings, blueprints, and technical information which are delivered to the Government under this Grant, and to authorize others to do so. With respect to data which are not originated during the effort Grantee shall give a similar license but only to the extent that Grantee and those in privity with Grantee have the right to give such license without paying compensation to others because of giving the license. At the time of giving or reporting any such data, Grantee shall make all reasonable effort to advise Grantor (1) of all invasions of the right of privacy contained therein and, (2) of all portions of such data copied from work not composed or produced during the effort and not licensed under this provision.

(c) Notwithstanding the provision of the preceding paragraph, the Grantee and the Government may agree that specifically designated data shall not be published for sale by the Government, nor shall the Government authorize others to do so when such data are published by the Grantee, and shall so refrain so long as the data are protected by copyright.

RESTRICTIONS ON PRINTING: Unless otherwise authorized in writing by the Grants Officer, reports submitted hereunder shall be reproduced only by duplicating processes and shall not exceed 5,000 single page reports or a total of 25,000 pages of a multiple-page report. To satisfy the requirement of the Defense Technical Information Center the copy of the technical report submitted to the Defense Technical Information Center must be black typing or reproduction of black on white paper or suitable for reproduction by photographic techniques. Reprints of published technical articles are not within the scope of this paragraph.

PUBLICATIONS:
1. Any publication resulting from work under this Grant shall contain the following on the title page or on the page immediately following the title page:

> This work relates to Department of Navy Grant N00014-91-J-1369 issued by the Office of Naval Research. The United States Government has a royalty-free license throughout the world in all copyrightable material contained herein.

2. Any transfer of copyright ownership in such publication will provide that the transfer of copyright ownership is subject to the U.S. Government's royalty-free license throughout the world in all copyrightable material contained in the publication.

CIVIL RIGHTS ACT: This Grant is subject to the compliance requirements of the "Civil Rights Act of 1964," 78 Stat. 241 (Public Law 88-352) relating to nondiscrimination in Federally assisted programs. The Grantee has signed an Assurance Compliance with the nondiscriminatory provisions of the Act.

FINANCIAL RECORDS AND REPORTS: The Grantee shall maintain adequate records to account for the expenditures made under this Grant and, when applicable to this document the actual amount of cost participation. Upon completion or revocation of this Grant, whichever occurs earlier, the Grantee shall furnish to the Grants Administrator, a Financial Status Report in accordance with OMB Circular A-110, Attachment G, Exhibit 1, showing a breakdown of expenditures made in performance of this Grant. Such financial statement may be on a cash or accrual basis depending upon the Grantee's accounting system. Such financial statement may be in the same detail as contained in the Grantee's approved budget for this Grant and shall be submitted no later than 90 days after the end of the annual reporting period or completion or revocation of the Grant. The Grantee's financial records are subject to audit by the Government when desired by the Grantor.

UNEXPENDED FUNDS AND EARNED INTEREST: After the end of the Grant period, any uncommitted funds and any interest earned by Grant funds on deposit shall be returned to the Office of Naval Research by check made payable to "Office of Naval Research."

RESTRICTION ON TRAVEL: The Grantee must obtain prior written approval from the Grants Officer, before funds provided under this Grant may be expended to provide for travel for persons from Communist Bloc countries.

TRAVEL BY GOVERNMENT EMPLOYEES: Funding of travel by employees of the U. S. Government with funds provided under this Grant is prohibited.

REVOCATION: This Grant may be revoked in whole or in part by the Grants Officer after consultation and agreement with the Grantee, provided that such revocation shall not affect any commitment which, in the judgment of the Grants Officer and the Grantee, has become firm prior to the effective date of the revocation; and funds not committed by the Grantee prior to the revocation shall be returned to the Office of Naval Research.

UNITED STATES OF AMERICA

FOR THE OFFICE OF NAVAL RESEARCH

BY: Gail D. Boger GAIL D. BOGER GRANTS OFFICER
(GRANTS OFFICER)

25 Feb 91
(DATE)

ATTACHMENT NUMBER 1

Distribution List for Reports

ADDRESSEE	NUMBER OF COPIES
Scientific Officer Code: 1133 Andre M. Van Tilborg Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217-5000	3 copies of proceedings
Grant Administrator Office of Naval Research Resident Representative N68883 Charles S. Draper Laboratory 555 Technology Square MS54 Cambridge, MA 02139-3539	1 copy of proceedings
Defense Technical Information Center Building 5, Cameron Station Alexandria, Virginia 22314	1 copy of proceedings